UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Design and implementation of a self-response tool for logistic platforms

Autor: Aleix Falgueras Casals

Director: Dr. Pau Fonseca i Casas

Tutor de GEP: Fernando Barrabes Naval

Fecha de la defensa: 27 de Abril de 2020

Especialidad: Ingeniería del Software

Titulación: Grado en Ingeniería Informática

Resumen

La logística es un conjunto de medios y métodos necesarios para llevar a cabo la organización de una empresa o de un servicio, especialmente de distribución. Su aplicación decide el futuro de las empresas e influye directamente en los beneficios. En este trabajo nos centramos en la logística de las empresas de distribución de mercancías, en concreto en las de productos alimenticios. De nada sirve tener el mejor género si no eres capaz de hacerlo llegar al cliente.

La constante innovación en informática conlleva tareas de actualización y adaptación. No obstante, parece que los sistemas logísticos se han quedado rezagados en lo que respecta a nuevas tecnologías. No se tiene en cuenta información externa a la hora de tomar decisiones y la gran mayoría de sistemas siguen usando protocolos de comunicación antiguos. Las estrategias de planificación actuales obvian factores clave, como las previsiones meteorológicas. La mala logística no solo conlleva una pérdida de beneficios; los alimentos que no llegan a su destino terminan desechados. Así pues, mediante una buena logística, no solo se incrementarían los ingresos, la sostenibilidad del proceso también mejoraría.

La aplicación que proponemos aprovecha los últimos avances tecnológicos para optimizar la distribución, el control de los productos y las estrategias de planificación. Queremos que esas mejoras se traduzcan en un incremento de ingresos y una disminución de desperdicios. Somos conscientes de que no podemos asegurar todas las entregas, pero creemos que hay un margen de mejora considerable.

Nuestra aplicación se sustenta sobre tres pilares: OPC UA (Object Linking and Embedding for Process Control Unified Architecture), IoT (Internet of Things) y servicios web. OPC UA nos permite comunicarnos con los sistemas ya existentes de una forma más segura y cómoda que con los protocolos OPC convencionales. El sistema IoT proporciona información constante sobre el estado actual de las mercancías, menos operarios y más dispositivos. Los servicios web permiten mejorar la respuesta y automatizar procesos, así como considerar los factores externos en la toma de decisiones.

Escogimos MongoDB como sistema de persistencia, una base de datos NoSQL. Una base de datos relacional se vería desbordada en cuanto a espacio y tiempo de respuesta. Al haber escogido MongoDB el diseño del modelo de datos será un factor clave en el rendimiento final. Trabajamos con tres colecciones: *product* para los productos, *route* para las rutas y *plan* para los planos de distribución generados.

Hemos diseñado un demo que permite ver el funcionamiento de la aplicación: ClimateFront. Simulamos un servidor OPC UA al que nos sincronizamos con un cliente OPC UA. La sincronización consiste en propagar la información del servidor a MongoDB a través de notificaciones. La información del servidor se guarda en nodos. El cliente percibe notificaciones de los nodos a los que está suscrito.

El servidor simula entradas de productos a un almacén. Informamos al cliente de la nueva cantidad recibida para cada producto vía notificación. Incrementamos la cantidad total de los productos en la colección product de MongoDB acorde a los nuevos datos.

El usuario puede crear rutas para un conjunto de los productos disponibles. Para cada ruta se indica la estrategia de planificación a seguir, aunque actualmente solo disponemos de una. El plano de las rutas es calculado usando las previsiones climatológicas obtenidas a través del servicio web OpenWeatherMap.

Paralelamente, emulamos lo que sería una planta de almacén usando FlexSim (software de simulación de eventos discretos 3D). Para poder realizar la simulación hace falta definir el modelo, el conjunto de elementos que generarán los eventos. ClimateFront se puede conectar al modelo vía OPC UA para proporcionar datos de entrada reales y mejorar las simulaciones. El módulo de estadísticas de FlexSim nos permite ver los puntos flojos del sistema y anticiparnos a los cuellos de botella.

No pudimos recrear el ecosistema IoT por falta de tiempo y medios, sin embargo, hemos cumplido la mayoría de los objetivos propuestos. Sentamos las bases a nivel arquitectónico para las futuras aplicaciones y demostramos que la distribución se puede, y se debe, hacer mejor. Combinar e integrar los nuevos avances con el software actual es la clave para marcar la diferencia en el sector. Toda la investigación realizada ha sido documentada. Esperamos que en un futuro alguien tome el relevo y termine las tareas pendientes para conseguir la aplicación de logística definitiva.

Índice

Índice de tablas

Índice de figuras

Introducción y contextualización

Vivimos en un mundo donde lo que hoy es nuevo mañana es viejo. El ritmo de crecimiento en el ámbito tecnológico es vertiginoso y hay que actualizarse constantemente para no quedarse atrás.

Uno de los últimos fenómenos emergentes es el denominado IoT (Internet of Things), que aplicado a la logística permite mejorar el control y estado de las mercancías. Queremos usar esta tecnología, entre otras, para desmarcarnos de protocolos de comunicación y estrategias antiguas como los sistemas cerrados OPC (Object Linking and Embedding for Process Control) y la falta de información externa en la toma de decisiones.

Nuestra intención con este proyecto es arrojar un poco de luz sobre la arquitectura OPC UA (Unified Architecture) y beneficiarnos de la información que ofrecen los servicios web para mejorar nuestra respuesta. Finalmente, optimizar todo el ciclo de distribución y mejorar el control de los productos a través del ecosistema IoT. El uso de dispositivos inalámbricos también nos permitirá automatizar procesos. Desarrollaremos una aplicación que integre todos esos conceptos, carentes a día de hoy en la mayoría de los sistemas de logística.

Para lograrlo analizaremos el entorno en el que nos movemos: mercado, actores, soluciones actuales, etc. y desarrollaremos un prototipo en sintonía, siempre poniendo el foco en la distribución de comestibles como producto principal. Sin embargo, nuestro objetivo no es crear un producto 100% funcional y listo para su uso comercial, sino sentar las bases de la arquitectura, estructura y patrones a seguir para futuras aplicaciones.

Toda la información recopilada será correctamente documentada para futuras investigaciones relacionadas con el transporte y la logística de inLab FIB.

Conceptos

IoT: Internet of Things

La definición de IoT podría ser la agrupación e interconexión de dispositivos y objetos a través de una red (bien sea privada o Internet, la red de redes), dónde todos ellos puedan ser visibles e interaccionar. Entendemos por dispositivo cualquier objeto que se pueda conectar a Internet e interaccionar con otros dispositivos sin necesidad de supervisión humana, logrando así una interacción M2M (Machine to Machine). (Gracia 2019)

En la Fig. 1 se muestra el típico ecosistema de una aplicación de logística basada en IoT. La idea es que todo está conectado y sincronizado en tiempo-real y de forma independiente.

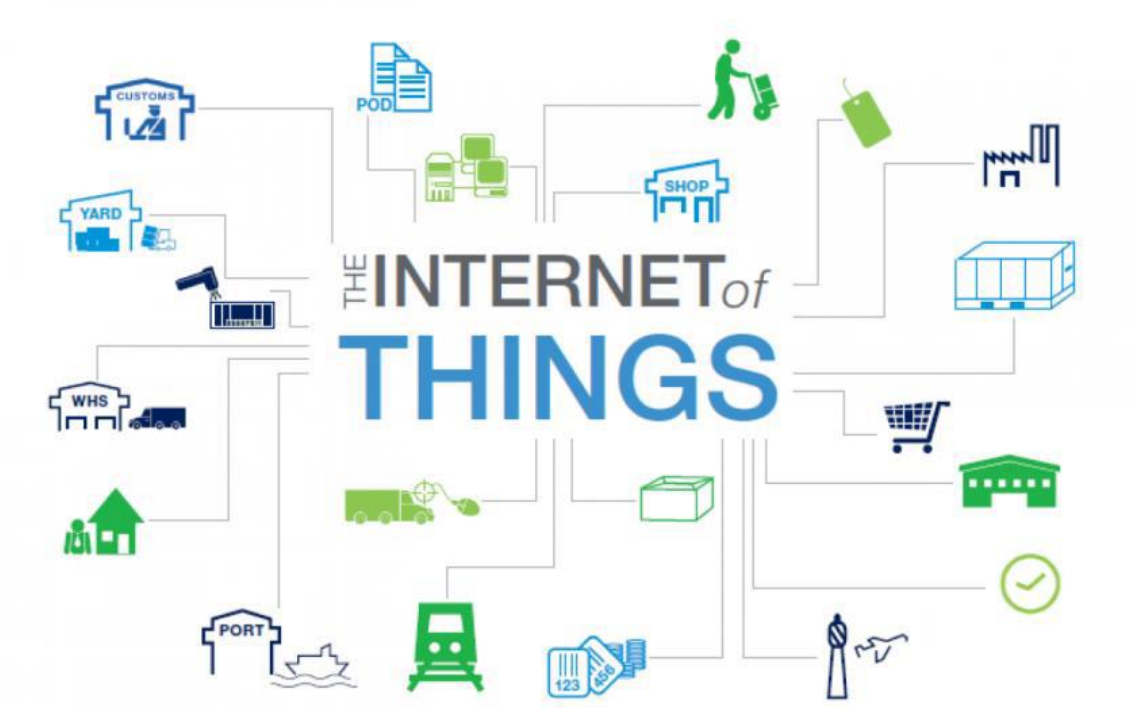

Fig. 1: Ecosistema basado en IoT[1]

[1] Fuente: https://medium.com/datadriveninvestor/the-internet-of-things-90263f7b1249 [consulta en: 18 Abril 2020]

OPC UA: Object Linking and Embedding for Process Control Unified Architecture

La OPC Foundation ofrece estándares para intercambio de datos en automatización industrial. Esto incluye la especificación OPC-DA (OPC - Data Acces) para datos actuales, así como OPC-A&E (OPC - Alarms and Events), para alarmas y eventos y OPC-HDA (OPC – Historical Data Acces) para datos históricos (ver "Anexo A: OPC" para más información).

Todas estas especificaciones están basadas en la tecnología COM/DCOM[2] venida a menos. Un primer paso hacia las tecnologías punta fue OPC XML-DA, que utilizaba XML sólo como formato de transporte de datos y por tanto no cumplía con los requisitos típicos de rendimiento en aplicaciones OPC.

Con la lección aprendida de OPC XML-DA, la OPC Foundation creó un nuevo estándar llamado Unified Architecture (OPC-UA). Aquí, el transporte puede realizarse utilizando servicios web que se manejan bien a través de cortafuegos con estándares como SOAP y HTTP, o un protocolo binario TCP optimizado para comunicaciones de altas prestaciones. OPC-UA ofrece comunicaciones entre aplicaciones inter-operativas, independientes de plataforma, de alto rendimiento, escalables, seguras y fiables.

Como vemos en la Fig. 2, el cambio de tecnología entre COM/DCOM de Microsoft a protocolos de transporte punteros e independientes-de-plataforma permiten a las aplicaciones OPC-UA ejecutarse en dispositivos inteligentes y controladores a la vez que en sistemas DCS (Distributed Control System) o SCADA[3] (Supervisory Control And Data Acquisition), hasta a niveles empresariales con sistemas MES (Manufacturing Execution Systems) y ERP (Enterprise Resource Planning). Esto aumenta inmensamente su rango de uso en comparación con aplicaciones clásicas de OPC.

Además del transporte, el segundo gran logro de OPC-UA es el modelo de información. OPC-UA ofrece modelos de información ricos y extensibles empleando conceptos orientados a objetos, permitiendo metadatos, así como datos complejos. (AGM – Larraioz Elektronika 2016)

Los mecanismos extensibles permiten definir modelos de información estándares a otras organizaciones que pueden utilizar la infraestructura de comunicación OPC-UA y enfocarse en estandarizar la información que será expuesta, que es justo lo que necesitamos para la integración de nuestra aplicación con las máquinas ya existentes.

[2] DCOM (Distributed Component Object Model) es una tecnología propietaria de Microsoft para desarrollar componentes de software distribuidos sobre varias computadoras y que se comunican entre sí. Extiende el modelo Component Object Model (COM) de Microsoft y proporciona el sustrato de comunicación entre la infraestructura del servidor de aplicaciones COM+ de Microsoft. (Wikipedia 2019)

[3] SCADA: Software que permite controlar y supervisar procesos industriales a distancia.

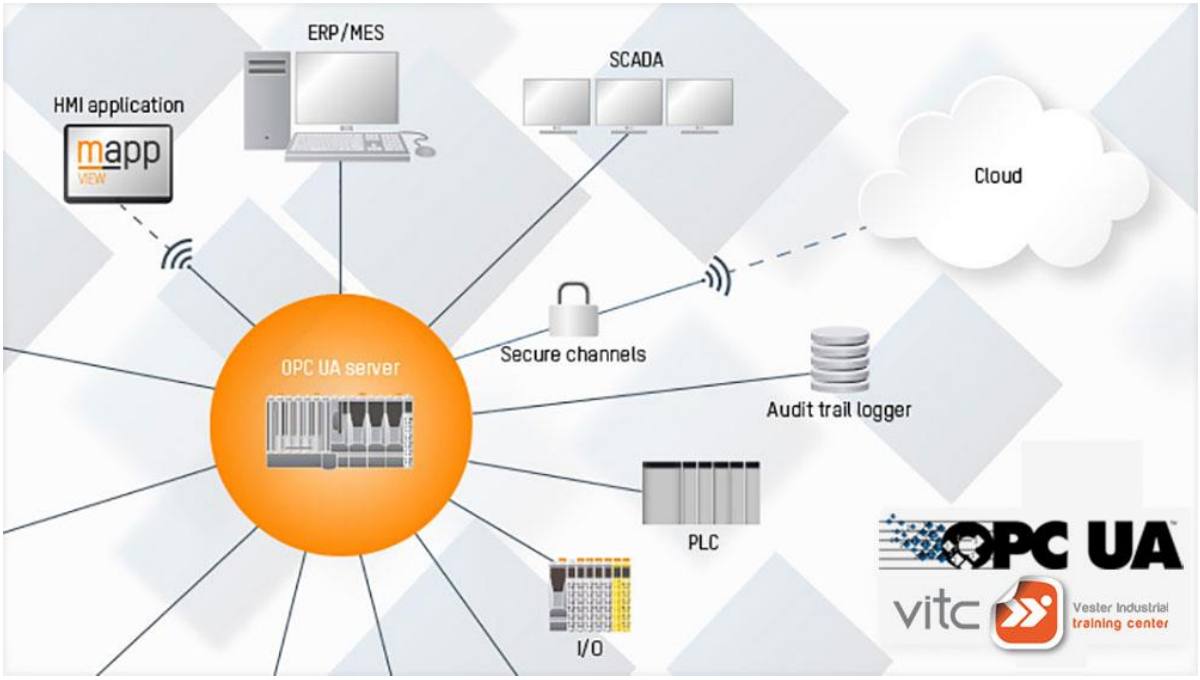

Fig. 2: Múltiples canales de comunicación de un servidor OPC UA[4]

Servicios web

Un Web Service, o Servicio Web, es un método de comunicación entre máquinas conectadas a una red, pública o privada. Es una colección de protocolos abiertos y estándares usados para intercambiar datos entre aplicaciones o sistemas. Las aplicaciones escritas en varios lenguajes de programación que funcionan en plataformas diferentes pueden utilizar *web services* para intercambiar información a través de una red. (García 2015)

Generalmente, la interacción se basa en el envío de solicitudes y respuestas entre un cliente y un servidor. El cliente solicita información y el servidor genera una respuesta que envía de vuelta al cliente, adjuntando otra serie de datos que forman parte de esa respuesta. Por tanto, podemos entender un servicio web como un tráfico de mensajes entre dos máquinas. (Lázaro 2018)

[4] Fuente: https://vestertraining.com/caracteristicas-opc-ua-scada/ [consulta en: 18 Abril 2020]

Problema a resolver

Nuestro propósito es mejorar la oferta actual de aplicaciones de logística de mercancías alimentarias en el mercado incorporando una serie de mejoras acorde al entorno en el que trabajamos.

El objetivo principal es diseñar, crear, un producto capaz de resolver/mejorar los siguientes puntos:

1. La comunicación de las aplicaciones de logística actuales con los sistemas industriales basados en OPC; se necesita una forma óptima, fiable y compleja para la transmisión de la información: OPC UA.

2. La falta de información de terceros en el sistema, lo cual provoca una serie de problemas que podrían ser evitables: la interrupción de una entrega por condiciones meteorológicas adversas, retrasos en las entregas debido a problemas políticos de un país, etc.

3. La toma de decisiones en tiempo-real de forma automática, hoy en día limitada por el único uso de la información de las máquinas conectadas vía Ethernet como fuente de entrada. Existen sistemas capaces de monitorizar las condiciones de humedad y temperatura de los paquetes que se están transportando y así controlar si el pedido se encuentra en un estado de conservación óptimo. (Alsina 2018)

Actores implicados

El producto resultante será una potente herramienta para las compañías que mueven miles de comestibles por todo el mundo, permitiendo optimizar su empresa y mejorar sus beneficios.

Aparte, los clientes verán reducidos los problemas causados por los proveedores como la falta de género o el mal estado del alimento y tendrán una mejor imagen de las marcas que consumen, lo que provocará una mayor fidelización y aumento del branding[5].

Y, por último, el medio ambiente y la sostenibilidad. El uso de nuestro sistema va a permitir un menor desperdicio de alimentos y ahorro energético, siendo fieles así al compromiso de la Universitat Politècnica de Catalunya con el planeta.

[5] Branding es un anglicismo empleado en mercadotecnia que hace referencia al proceso de hacer y construir una marca mediante la administración estratégica del conjunto total de activos vinculados en forma directa o indirecta al nombre y/o símbolo (logotipo) que identifican a la marca influyendo en el valor de la marca, tanto para el cliente como para la empresa propietaria de la marca. (Wikipedia 2018)

Justificación

Hemos decidido apostar por una solución nueva y rompedora tanto a nivel teórico como tecnológico.

Desde el punto de vista teórico, como vemos en (Estrada-Moreno et al. 2018), prácticamente todos los algoritmos de las aplicaciones de logística que deciden la asignación de un cliente a un depósito/almacén y optimizan las rutas de reparto parten de la premisa de que el problema combinacional[6] que se presenta no contiene ningún componente de incertidumbre. La consecuencia es la falta de una solución óptima cuando se presenta dicha incertidumbre.

Siguiendo las pautas del algoritmo (Estrada-Moreno et al. 2018) en nuestra solución el problema de distribución se modela como un problema de enrutamiento de vehículos de depósito múltiple (VRP multi-depot) con tiempos de viaje estocásticos. Estos tiempos de viaje no solo son de naturaleza estocástica, sino que la distribución de probabilidad específica utilizada para modelarlos depende de las condiciones climáticas particulares del día de entrega.

El algoritmo que implementaremos aborda el desafío de resolver la versión estocástica del VRP de depósito múltiple con el uso de un heurístico. Un heurístico es la combinación de alguna metaheurística (estrategias que guían el proceso de búsqueda) y un muestreo o validación, realizado con simulación, para determinar el espacio de estados a explorar, con el objetivo de resolver un problema de optimización combinatoria (COP) estocástico. Aunque existen métodos exactos para dichos problemas, estos solo son viables para pequeñas instancias. Debido a la complejidad que se presenta se ven desbordados cuando los datos de entrada son de un tamaño superior al previsto, lo que hace necesario un enfoque basado en heurísticas.

El objetivo principal es determinar el plan de asignación de cliente a depósito y el plan de enrutamiento posterior que minimice el costo total esperado de la actividad de distribución. Este proceso de optimización debe tener en cuenta las restricciones sobre la capacidad de cada vehículo (la misma para todos), así como sobre la capacidad de cada depósito (distintas entre sí y determinada por el número de vehículos asociados a este).

Aquí es donde nos desmarcamos de los demás modelos. Aunque ha habido algunos estudios previos que han abordado versiones estocásticas del VRP de múltiples depósitos, ninguno de los trabajos previos ha considerado depósitos capacitados (almacenes de tamaño variable), que es el escenario realista que este algoritmo tiene en cuenta.

Y desde el punto de vista tecnológico, plantemos una arquitectura basada en OPC UA, que aunque ya hay aplicaciones que tienen un sistema basado en estos estándares (OPC Fundation 2017) ninguna de ellas es pública/comercial.

[6] El diseño de rutas de distribución efectivas generalmente se puede modelar como un problema de optimización combinatoria.

Solución propuesta

Vistos los problemas, creemos que la mejor solución es diseñar una aplicación de auto respuesta que se encargue de gestionarlo todo.

Para poder realizar todas las funciones la aplicación recibirá información de tres canales distintos:

- A través del software ya existente se le comunicará los productos y activos de la empresa.
- A través de servicios web de terceros se le informará de la climatología, incidencias en aeropuertos, congestiones de tráfico, etc.
- A través de un conjunto heterogéneo de dispositivos inalámbricos recibirá datos del mundo real.

Para recabar la información del software existente será necesario realizar una integración OPC UA con las máquinas y sistemas que cumplan los estándares OPC.

Para contactar con los servicios web primero deberemos decidir qué queremos que nos aporten y cuales usaremos de todos los que ofrece el mercado.

Y por último, deberemos montar nuestro sistema en tiempo-real basado en IoT, que nos deberá permitir controlar los siguientes aspectos:

- Información del producto.
- Control energético.
- Gestión del tiempo.
- Control de temperatura.
- Evolución del envío.
- …

Para optimizar las rutas de envíos y asignaciones de clientes a almacenes implementaremos el heurístico comentado en el apartado anterior, que para calcular el coste de los viajes tiene en cuenta el factor climático en la función congestión del tráfico y lo que conlleva dicha aleatoriedad de eventos. Ver “Anexo B: VRP de múltiples depósitos con tiempos de viaje estocásticos, aproximación heurística” para más información.

Estos eventos son modelados a partir de una distribución de probabilidad log-normal. La razón principal es que esta familia de distribución permite modelar datos sesgados positivamente, lo cual es una característica importante a tener en cuenta (Estrada-Moreno et al. 2018).

La adaptación a los impactos relacionados con el clima de las redes de transporte terrestres se identifica como una **estrategia clave para minimizar las consecuencias del cambio climático**. Modificar la logística de acuerdo con las advertencias meteorológicas proporcionadas por las agencias meteorológicas parece una forma realista y práctica de adaptación al cambio climático (Estrada-Moreno et al. 2018). Consultar “Anexo C: El caso irlandés” para más información).

Alcance

Al ser un proyecto con una doble finalidad: crear un producto e investigar un campo, no es posible detallar el abasto total a día de hoy. Este dependerá del enfoque que vaya adoptando el proyecto a medida que avanzamos, sin embargo, nos es posible comentar a grandes rasgos nuestra hoja de ruta.

Como ya hemos expuesto anteriormente, básicamente perseguimos dos objetivos:

1. Crear una aplicación para la logística y transporte de alimentos que reciba información de dos canales distintos: OPC UA y servicios web, y que sea capaz de responder automáticamente a ciertos estímulos en función de sus conocimientos.
2. Documentar todo el proceso, investigar los distintos campos aplicados en el área de la logística y sentar las bases para el desarrollo de aplicaciones relacionadas.

Si nos centramos más en investigar y documentar o en construir el producto dependerá de cómo se vayan desarrollando los acontecimientos y los distintos inconvenientes que nos encontremos.

Objetivos

- Diseño de la arquitectura del sistema y el modelo de datos.
- Diseño de la comunicación entre los diferentes módulos.
- Diseño de la interfaz gráfica de usuario a través de un frontal web.
- Implementación del sistema.

Subobjetivos

- Investigar cómo aplicar IoT a la logística, los servicios web necesarios para satisfacer nuestros requisitos y cómo implementar una especificación OPA UA.
- Documentar todo el proceso de investigación en IoT, servicios web e integración con OPC UA.
- Sentar las bases de la arquitectura y patrones a seguir para futuras aplicaciones de logística y transporte.

Otros requerimientos funcionales o no funcionales

Al ser un proyecto ambicioso y tener un gran componente de investigación es muy probable que vayan saliendo requisitos no funcionales a medida que avancemos y vayamos definiendo la idea final. Este es uno de los principales motivos por el que hemos escogido Scrum como metodología.

Los requisitos no funcionales pueden ir desde el tiempo de respuesta del sistema IoT o servicios web hasta la capacidad de banda ancha o consumo energético de toda la organización. Dependerá también de a qué cliente final se quiere enfocar el producto y sus necesidades concretas.

Y como requisitos funcionales, es posible que detectemos alguna característica/mejora que no hayamos tenido en cuenta para el sector en cuanto profundizamos más en el mundo de logística y transporte alimenticio. Aunque a priori tenemos claras las funcionalidades que queremos que tenga nuestra aplicación.

Obstáculos y riesgos

Como obstáculo y riesgo principal remarcaríamos la obtención y fabricación de datos con los que poner a prueba nuestro sistema. Tendremos que definir exactamente de donde sacar la información o como poder simularla de la forma más fiel posible a la realidad para realizar las pruebas correctamente.

La falta de conocimiento y documentación del entorno también serán obstáculos a tener en cuenta, es muy probable que afecten a los tiempos planificados inicialmente e incluso modifiquen algún objetivo.

Metodología

Usaremos metodología Agile, concretamente Scrum, que permite tener resultados desde prácticamente el primer día, facilitar la replanifiación y detectar problemas antes de tiempo.

En Scrum se realizan entregas parciales y regulares del producto final, priorizadas por el beneficio que aportan al receptor del proyecto. Por ello, es especialmente indicado para proyectos en entornos complejos, donde se necesita obtener resultados pronto, donde los requisitos son cambiantes o poco definidos, donde la innovación, la competitividad, la flexibilidad y la productividad son fundamentales (Albaladejo 2018).

La metodología se basa en el desarrollo de iteraciones/sprints y entregas, partiendo de una lista de objetivos/requerimientos priorizados por el cliente (Product Owner). En cada iteración se decide junto a este cuáles serán los objetivos a cumplir y el equipo lista las tareas necesarias para ello.

Al final de cada sprint se realiza una revisión donde se presentan al cliente los requisitos completados. En función de los resultados mostrados y de los cambios que haya habido en el contexto del proyecto el cliente realiza las adaptaciones necesarias. También se hace una retrospectiva donde el equipo analiza cómo ha sido su manera de trabajar y cuáles son los problemas que podrían impedirle progresar adecuadamente.

En nuestro caso mi director de TFG, el Dr. Pau Fonseca, será el *Product Owner* y yo asumiré el rol de desarrollador y Scrum Master. La Fig. 3 ilustra todo el proceso.

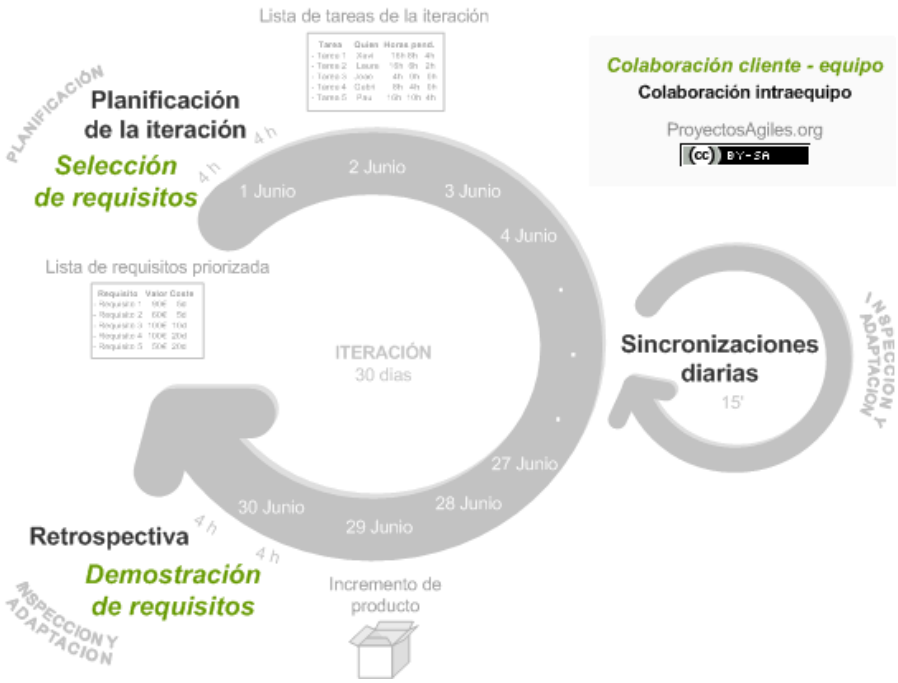

Fig. 3: Proceso Scrum[7]

[7] https://proyectosagiles.org/que-es-scrum/ [consulta en: 18 Abril 2020]

Planificación

El proyecto se inició el 16 de setiembre de 2019 y su defensa está prevista el 27 de Abril de 2020.

La duración es más larga de lo normal debido a una prórroga de 4 meses. Dicha prorroga no ha conllevado un aumento en las horas de dedicación. La planificación la realizamos como si se fuese a presentar el día original, el 28 de enero de 2020. Aunque el día de presentación ha cambiado, el orden de procedencia y las horas planificadas se ha mantenido igual.

Actualmente el TFG (trabajo de fin de grado) equivale a 18 créditos y cada crédito tiene una carga de 30 horas, con lo que se dispone de 540 horas para todo su desarrollo.

Recursos

Por lo que respecta a recursos humanos, solo estaré yo: Aleix Falgueras Casals.

Identificamos dos tipos de recursos materiales: recursos hardware y recursos software.

Recursos hardware:

- **R1**: Portátil Lenovo Ideapad S540-14IWL

Recursos software:

- **R2**: Git-hub: Plataforma de repositorios online.
- **R3**: Trello: Herramienta para el control de las tareas.
- **R4**: Python 3.7.6: Librerías de Python.
- **R5**: PyCharm: Editor de código Python.
- **R6**: Mongo DB: Base de datos.
- **R7**: Robo 3T: GUI de MongoDB.
- **R8**: UaExpert: Cliente OPC UA.
- **R9**: Insomnia: Cliente API REST.
- **R10**: FlexSim: Simulación de procesos.

Descripción de las tareas

1. Situación actual del proyecto

Primera toma de contacto con el proyecto y el mundo OPC/OPC UA, investigación de la situación actual de las empresas y estado de aplicación de los estándares OPC UA.

2. Gestión del proyecto

Tareas relacionadas con la gestión y documentación del proyecto.

2a Contexto

2b Justificación

2c Alcance

2d Metodología

2e Planificación

2f Presupuesto

2g Sostenibilidad

2h Redacción final de la memoria

2i Diseñar presentación y preparar exposición oral

2j Reuniones de control y seguimiento con el tutor.

3. Búsqueda de tecnologías a usar

Elección del lenguaje de programación y herramientas necesarias para el desarrollo.

4. Instalación y documentación de Python

Instalación de la versión de Python, IDE y primeras pruebas.

5. Instalación y documentación de MongoDB

Instalación de MongoDB, Robo 3T y primeras pruebas.

6. Diseño de la base de datos de Mongo

Diseño del modelo de datos y entidades con las que se va a trabajar.

7. Implementación del sistema

7a Módulo de mongo

Implementar módulo de conexión a Mongo DB.

7b Módulo OPC UA

Implementar cliente y servidor OPC UA y la comunicación con los dispositivos IoT.

7c API REST

Implementar API REST con la lógica del negocio.

7d Web Service climatológico

Implementar los protocolos de comunicación con un servicio web climatológico.

7e Frontal web

Implementar frontal web para la presentación de la información.

8. Creación del entorno de pruebas

Instalación de las herramientas necesarias para ir probando el proyecto y creación de los distintos juegos de prueba.

8a Instalación y documentación de UaExpert

8b Testear el servidor con UaExpert

8c Instalación y documentación de Insomnia

8d Testear la API REST con Insomnia

9. Simulación con FlexSim

9a WorkShop con FlexSim

Primera toma de contacto de FlexSim y primeros ejercicios prácticos.

9b Creación del modelo FlexSim para el proyecto

Fases del proyecto

Las tareas se agrupan por fases de desarrollo: [*nombre*: *código*]

- Planificación: **PLA**
- Documentación: **DOC**
- Diseño e implementación del sistema: **IMP**
- Simulación: **SIM**
- Cierre del proyecto: **FIN**
- Universal (la tarea se desarrolla a lo largo de todo el proyecto): **UNI**

Consultar Tabla 1 para ver la relación entre fase y tarea.

Tabla resumen tareas

Tabla 1: Resumen tareas

Fase	Código	Tarea	Dependencias temporales	Recursos materiales	Horas
PLA	**1**	**Situación actual del proyecto**	**-**		**20**
-	**2**	**Gestión del proyecto**	1		**150**
PLA	2a	Contexto			30
PLA	2b	Justificación			15
PLA	2c	Alcance			5
PLA	2d	Metodología			5
PLA	2e	Planificación			15
PLA	2f	Presupuesto			15
PLA	2g	Sostenibilidad			5
FIN	2h	Redacción final de la memoria			40
FIN	2i	Diseñar presentación y preparar exposición oral			10
UNI	2j	Reuniones de control y seguimiento con el tutor			10
DOC	**3**	**Búsqueda de tecnologías a usar**	1, 2		**20**
DOC	**4**	**Instalación y documentación de Python**	3	R4,R5	**15**
DOC	**5**	**Instalación y documentación de MongoDB**	3	R6, R7	**15**
IMP	**6**	**Diseño de la base de datos de Mongo**	5	R6, R7	**30**
IMP	**7**	**Implementación del sistema**	4, 6	R2, R3, R4, R5, R6, 57	**220**
	7a	Módulo de mongo			30
	7b	Módulo OPC UA			80
	7c	API REST			50
	7d	Web Service climatológico			30
	7e	Frontal web			30
IMP	**8**	**Creación del entorno de pruebas**	7	R8, R9	**30**
	8a	Instalación y documentación de UaExpert			5
	8b	Testear el servidor con UaExpert			15
	8c	Instalación y documentación de Insomnia			5
	8d	Testear la API REST con Insomnia			5
SIM	**9**	**Simulación con FlexSim**	7	R10	**40**
	9a	WorkShop con FlexSim			15
	9b	Creación del modelo FlexSim para el proyecto			25
				Total:	**540**

Se omite mencionar el recurso material R1 ya que siempre es necesario.

La dependencia temporal implica que la tarea debe realizarse antes o paralelamente a la tarea indicada.

Gran parte de la investigación y documentación se realizó en la tarea "2. Gestión del proyecto", por eso tiene una cantidad de horas superior a lo esperado. Se han sobreestimado todas las tareas para poder manejar los contratiempos sin superar las 540 horas.

Diagrama de Gantt

Aunque hemos optado por Scrum, los sprints solo se reflejan en la fase de diseño e implementación del sistema (IMP), no obstante todo el proyecto seguirá un proceso iterativo. Los sprints tendrán una duración de 56 horas y 14 días, 4 horas de dedicación de media diaria.

En total se realizarán 5 sprints para cubrir toda la fase de IMP en 2 meses y medio, como se muestra el diagrama de Gantt de la Fig. 4.

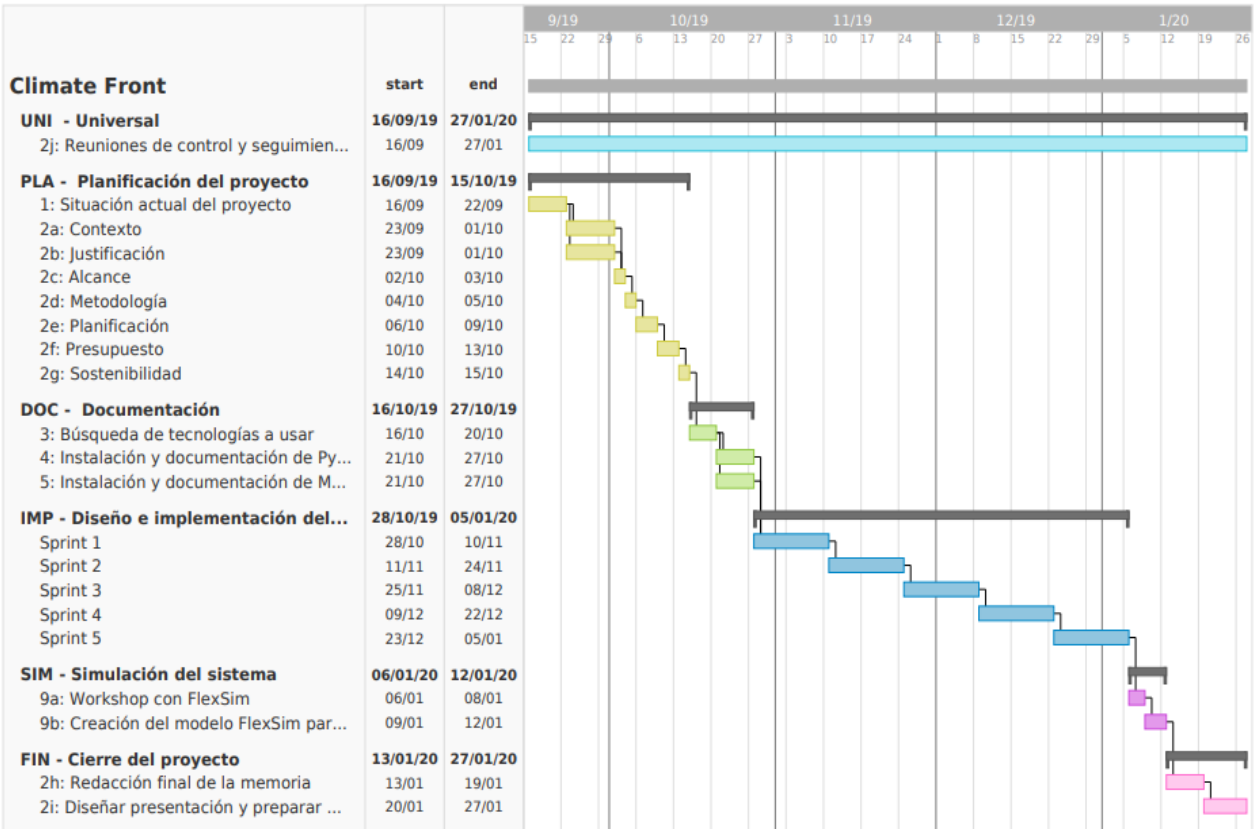

Fig. 4: Diagrama de Gantt[8]

[8] Fuente: https://www.teamgantt.com/ [consulta en: 18 Abril 2020]

Gestión del riesgo: Planes alternativos y obstáculos

Como comentamos anteriormente, el principal obstáculo será la obtención de datos para simular nuestra aplicación. A esto hay que sumarle las dificultades que surgirán al desarrollar con tecnologías punteras, librerías públicas y la posible falta de documentación.

En caso de ser necesario, se pospondrá la implantación del algoritmo de planificación y la implementación del frontal web. Las horas destinadas a estas tareas se reasignarán a otras más prioritarias. La implementación del algoritmo de planificación no debe suponer una gran pérdida de tiempo ya que el código en sí no es un objetivo prioritario (sí lo es su investigación y documentación), al igual que el frontal web, una tarea trivial pero costosa en tiempo.

Los protagonistas de este proyecto son: OPC UA, el diseño arquitectónico de la solución propuesta y la investigación de servicios externos que se podrían usar para mejorar la respuesta automática de la aplicación, y así se va a reflejar.

Gestión económica

Los costes se han calculado sin tener en cuenta la prórroga, acorde con la planificación.

Costes directos

Recursos humanos

Aunque estoy yo solo, identifico los distintos roles que he asumido a lo largo de todo el desarrollo.

Tabla 2: Costes recursos humanos

Rol	Tiempo (h)	Salario bruto (€/h)	Salario bruto estimado (€)
Director	200	20	4000
Analista	70	15	1050
Programador	220	12	2640
Tester	50	12	600
Total	**540**		**8290**

No se incluyen gastos en seguridad social ni impuestos ya que dependen de la regulación actual del país donde se desarrolle el proyecto.

Recursos hardware

El único recurso hardware empleado ha sido mi ordenador personal, un Lenovo S540-14IWL.

Fórmula para calcular la amortización:

$$Amortización = \frac{Precio\ de\ compra\ del\ equipo\ (€)}{Vida\ útil\ (años) \cdot Días\ laborables\ (días) \cdot Horas\ laborables\ al\ día\ (h)} \cdot Horas\ de\ uso\ (h)$$

(Eq. 1)[9]

[9] Días laborables = 250, horas laborables al día = 4 y horas de uso = 540.

Tabla 3: Costes recursos hardware

Recurso	Precio (€)	Unidades	Vida útil (años)	Amortización (€) (Eq.1)
Lenovo Ideapad S540-14IWL	770	1	4	103,95
Total				**103,95**

Recursos software

Ningún coste ya que todo el software es open-source o está bajo la licencia de estudiante de la UPC.

Resumen costes directos

Tabla 4: Resumen costes directos

Coste directo	Precio (€)
Recursos humanos	8290
Recursos hardware	103,95
Recursos software	0
Total	8393,95

Costes indirectos

Todo el proyecto se he realizado desde mi casa, a continuación se detallan los gastos aproximados al mes:

- Alquiler: 950 €
- Electricidad: 15 €
- Internet: 50 €
- Otros (material, mobiliario, etc.): 10 €

La suma da un total de 1015 € al mes, el TFG tiene una duración de 4 meses, con lo que la cifra final asciende a 1015 * 4 = **4060 €**.

Imprevistos

Para anticiparnos a los posibles retrasaos y contratiempos se reserva el **15%** de más de las horas iniciales previstas para el equipo:

Tabla 5: Costes imprevistos

Rol	Tiempo inicial (h)	Tiempo extra 15% (h)	Salario bruto (€/h)	Salario bruto estimado (€)
Director	200	30	20	600
Analista	70	11	15	165
Programador	220	33	12	396
Tester	50	8	12	92
Total	**540**	**82**		**1253**

Para los recursos hardware supondremos que hay un 5% de probabilidad de que el ordenador se estropee, el 5% del precio del ordenador de770 € es 115.5 €.

Total final: 115.5 + 1253 = **1368.5 €.**

Contingencia

Respecto a la contingencia, mismo enfoque que en los imprevistos, reservamos el **15%** de más en los costes directos e indirectos.

Tabla 6: Costes Contingencia

Tipo	Coste inicial (€)	Coste extra 15% (€)
Costes directos	8393,95	1259,09
Costes indirectos	4060	609
Total		**1868.09**

Presupuesto

Poniendo todo en conjunto nos sale el siguiente presupuesto:

Tabla 7: Presupuesto

Concepto	Coste (€)
Costes directos	8393,95
Costes indirectos	4060
Imprevistos	1368.5
Contingencia	1868.09
Total	**15.690,54**

Control de gestión

Al final de cada fase se revisarán todos los costes para ir controlando la evolución del proyecto.

En caso de que sea necesario se usarán los fondos reservados de imprevistos y contingencia para compensar cualquier problema que surja.

CE = Coste estimado.

CR = Coste real.

CHR = Consumo de horas reales.

CL = Consumo real.

CHE = Coste de horas estimado.

Indicador recursos humanos

Calcular está desviación nos hará conscientes de si estamos cumpliendo con la planificación y nos permitirá poder tomar decisiones a tiempo en caso contrario.

$$Desviación = (CHE - CHR) \cdot CE$$

Indicador costes hardware y software

Para calcular la desviación debida a costes de equipos o licencias no contemplados de inicio.

$$Desviación = CR \cdot CL$$

Indicador costes indirectos

Desviaciones entre el consumo aproximado y el real.

$$Desviación = (CE - CL) \cdot CR$$

Sostenibilidad

A continuación, comentamos la sostenibilidad del proyecto desde tres dimensiones: económica, social y ambiental.

Dimensión económica

Poniendo en contexto el presupuesto de las grandes empresas creo que este producto sería una buena inversión, no solo venderían más alimentos gracias a las múltiples mejoras en todos los procesos de la distribución sino que además reducirían gran parte de las pérdidas que sufren a día de hoy.

Sin embargo, al precio final de 16.000 euros aproximadamente habría que sumarle el coste de las licencias necesarias para poder vender la aplicación y el de los dispositivos IoT que se decidiera incorporar, lo cual podría incluso doblar el precio inicial.

Respecto a los costes de mantenimiento, habría que ver el producto final para poder estimarlo, como en el precio de venta dependerá de los dispositivos IoT que se decida incorporar y la vida útil de estos.

Dimensión social

A nivel personal, he visto incrementarse notablemente mis habilidades ya que ni conocía el lenguaje, ni las librerías ni la base de datos con la que he estado trabajando. El haber atravesado problemas personales durante la realización me ha hecho mejorar mi capacidad de reacción y toma de decisiones.

De cara al cliente, la aplicación va a permitir reducir el número de operarios encargados del control de las existencias, reduciendo así los costes de personal al contar solo con un pequeño equipo encargado del mantenimiento del sistema.

Y para el resto de la sociedad, se verán reducidos los problemas causados por los proveedores como la falta de género o el mal estado del alimento, lo que aumentará la satisfacción de los clientes de la empresa y con ello el consumo.

Dimensión ambiental

Desde una perspectiva interna, misma consideración que en la dimensión económica, en función de los dispositivos IoT finales que se incorporasen estaríamos hablando de una huella ambiental u otra, ya que estos dispositivos generalmente no tienen una vida útil muy larga. En cualquier caso, en este aspecto es casi seguro que no vamos a destacar positivamente, ya que los proyectos IoT no brillan por su sostenibilidad, no obstante, la historia es otra si cambiamos el enfoque.

Desde una perspectiva externa, nuestro producto aportará grandes mejoras ambientales al evitar el desperdicio de miles de alimentos que no llegan su destino o lo hacen en mal estado. No solo se reducirá el desperdicio de alimentos, el ecosistema IoT permitirá detectar problemas relacionados con el mantenimiento de los productos mejorando así la conservación y evitando que se malogren por cosas triviales, como, por ejemplo, la temperatura de un almacén.

Decisiones tomadas, problemas encontrados y solución final

Decisiones

Persistencia

Aunque en un inicio se contempló Sentilo como sistema de persistencia, al final hemos decidido trabajar con MongoDB (mongo, en adelante) por su creciente popularidad y el alcance actual del proyecto. Por el momento no hace falta un sistema tan completo y robusto como lo es Sentilo. Otra opción válida hubiese sido Elastic Search o Cassandra.

El requisito indispensable es que sea NoSQL debido al gran volumen de datos con los que vamos a trabajar, la escalabilidad que se prevé y la velocidad de respuesta necesaria. Con MongoDB, al ser una base de datos distribuida en su núcleo, la alta disponibilidad, la escalabilidad horizontal y la distribución geográfica están integradas y son fáciles de usar. Se decide usar Robo 3T como interfaz gráfica por su sencillez.

Lenguaje y librerías

Respecto al lenguaje de programación, nos hemos decantado por Python por su simplicidad y popularidad en el ecosistema IoT. C++ también hubiese sido una buena elección por como gestiona la memoria (más eficiente que Python), aunque personalmente me interesa más aprender Python. Se descartó Java por ser necesario tener la máquina virtual para la ejecución del código, lo que conlleva una pérdida de espacio considerable para los dispositivos IoT como los sensores o controles con los que se va a trabajar.

Librearías usadas (paquetes[10] en Python):

- **pymongo**: Driver python de conexión a MongoDB, como dice la web oficial de mongo: *PyMongo es la forma recomendada de trabajar con MongoDB desde Python.*
- **opcua**: Paquete del ecosistema FreeOpcUa, un proyecto open-source que incluye todo tipo de herramientas para el entorno de OPC UA, nos permite levantar un servidor OPC UA e interactuar con él a través de un cliente OPC UA. Se ha escogido FreeOpcUa por ser el proyecto más popular y completo en GitHub.

[10] Las versiones y links de documentación se pueden encontrar en fichero READ.ME del proyecto python adjunto.

- **Flask**: Framework que nos permitirá crear nuestra API REST de forma agilizada.
- **requests**: Librería para el manejo de peticiones y respuestas HTTP.

PyCharm como entorno de desarrollo por tener acceso a la versión completa gracias a la licencia de estudiante.

Simulación de un servidor OPC UA

Viendo la poca oferta en el mercado de softwares de simulación de servidores OPC UA he decidido implementar uno yo mismo con el paquete de *opcua*. De esta forma, no solo aprendo más sobre este mundo, sino que también puedo configurar todos los parámetros de la simulación y obtención de datos. En el futuro también se podrá usar como una primera aproximación de cómo implementar un servidor OPC UA en dispositivos IoT.

OpenWeatherMap

Escogimos este servicio para obtener la información meteorológica por su popularidad, facilidad de uso y ejemplos online. Aparte, la estructura JSON de la respuesta encaja perfectamente con nuestra base de datos y nos permite acceder directamente a la información vía clave-valor. Su versión gratuita ofrece previsión de cinco días en períodos de tres horas para una localidad.

Digital twin

Un Digital Twin o gemelo digital es una réplica virtual de un sistema que simula su comportamiento real con el fin de monitorizarlo para analizar su respuesta en determinadas situaciones y mejorar su eficacia.

Un gemelo digital, como modelo de simulación consta del modelo, la formalización que define su comportamiento y su codificación. Aquí se puede ver la formalización de un gemelo digital para el ámbito de la edificación u hospitalario (Fonseca i Casas, P.; Fonseca i Casas, A.; Garrido-Soriano, N.; Casanovas 2014) (Leiva, J.; Fonseca i Casas, P.; Ocana 2018). En el trabajo no desarrollaremos un modelo conceptual del gemelo digital por falta de tiempo y nos centraremos en la definición de un pequeño modelo de prueba codificado en FlexSim para el testeo de la aplicación.

Decidimos usar FlexSim como herramienta para codificar el *Digital Twin* porque a día de hoy es uno de los mejores softwares de simulación del mercado y porque ofrecen una versión gratuita muy potente. La experiencia en el workshop de FlexSim realizado en la UPC durante la elaboración de este TFG ha acelerado todo el proceso de aprendizaje.

Alcance final

Como comentamos en la introducción del apartado “Alcance” nos fue muy difícil decidir a priori el alcance final debido al componente de investigación y la ambición del proyecto.

Al final hemos terminado dejando de lado el módulo de IoT, no hemos indagado en los dispositivos que se podrían usar ni en la organización dentro del sistema por falta de tiempo y recursos. No obstante, como he dicho antes, el servidor OPC UA simulado se podrá usar como primera aproximación al servidor que tendrán que tener estos dispositivos.

Problemas

En el apartado anterior, *“*Obstáculos y riesgos*”*, expusimos que los principales problemas serían “la obtención y fabricación de datos con los que poner a prueba nuestro sistema”, y no íbamos desencaminados. Sin embargo, surgió otro problema inesperado que al final término afectando en gran medida al desarrollo y toma de decisiones del proyecto: la planificación.

La no previsión de que algún aspecto organizativo también podría afectar al progreso se debe a que solo contemplamos futuros problemas en el ámbito técnico, un error que no volveremos a cometer.

Planificación: Plan B

Debido a problemas personales y laborales tuve que solicitar una prórroga y presentar el TFG al siguiente cuadrimestre. Esta prórroga afecto a la planificación y al alcance inicial. A pesar de eso, si seguimos la filosofía Agile. En cada iteración avanzamos en los distintos módulos y al final de esta obteníamos una versión funcional mejorada del producto. Las tareas y el orden de ejecución eran reconsiderados al final de cada sprint teniendo en cuenta las prioridades y el resultado de la última ronda.

Hemos dejado de lado Trello ya que debido a la velocidad de desarrollo y al estar yo solo programando tampoco lo veía de mucha utilidad el ir moviendo tareas de un sitio a otro, para eso ya está el magnífico Bloc de notas.

FreeOPCUA: ¿Documentación? “We don’t do that here”

El que ha sido sin duda el principal problema y desafío de esta empresa: la falta de documentación para el paquete *opcua*.

Viendo la situación, cambiamos el enfoque y nos lo tomamos más como un reto. Al final, a base de prueba y error conseguimos terminar todo el módulo e implementar las funcionalidades que queríamos.

Es lo que tiene trabajar con nuevas tecnologías y proyectos open-source, que se hace lo que se puede y sin ánimo de lucro, con lo que no se puede exigir mucho sino agradecer el esfuerzo de la comunidad.

FlexSim: No todo son buenas noticias

El workshop realizado en la UPC fue un gran éxito y el docente hizo que el programa pareciese simple (que no lo es). Después del cursillo exprés de tres días pudimos desarrollar una buena simulación de lo que sería la planta de un almacén sin problemas.

El problema surgió en la recta final, cuando había que conectar el modelo con el servidor OPC UA para que este proporcionara datos de entrada a la simulación. No fue muy complicado encontrar la forma de hacerlo. Sin embargo, una vez que la conexión estaba definida y queríamos explorar el servidor para ver qué variables usar, saltaba un error de conexión.

El error en sí no tiene mucho sentido, ya que si testeas la conexión desde FlexSim te dice que es correcta. A través del log de nuestro servidor pudimos ver que efectivamente recibíamos una petición de conexión y se aprobaba.

Nuestra hipótesis es que el problema se encuentra en la forma interna en la que FlexSim manda las peticiones al servidor. Nos basamos en que no ha habido problema alguno al lanzar peticiones y obtener respuesta ni desde el cliente OPC UA programado ni desde UaExpert, un software que emula un cliente OPC UA.

Hemos contactado con el soporte de FlexSim por correo indicándoles el problema y a día de hoy siguen sin saber el porqué. En su defensa hay que decir que este módulo de conexión OPC UA acaba de salir en la última versión del producto y que el código empleado es open-source (no tiene que seguir los estándares a raja tabla ni está regulado por ninguna entidad superior).

Solución final

Como hemos visto anteriormente, la idea inicial era desarrollar una aplicación que se comunicara con dispositivos vía OPC UA, recibiera información externa a través de servicios web y dispositivos IoT y tomará decisiones en tiempo real.

Estas decisiones consistirían en planificar rutas para los distintos productos de un stock usando el algoritmo de enrutamiento de vehículos de depósito múltiple (VRP multi-depot) con tiempos de viaje estocásticos y teniendo en cuenta eventos externos. Obligatoriamente las condiciones meteorológicas, pero pudiendo añadir más información recibida a través de los dispositivos IoT u otros servicios web en un futuro.

Al final, decidimos reducir el alcance eliminando el módulo de IoT y el sistema propuesto fue traducido a una serie de objetivos. Los hemos cumplido todos, aunque el frontal web ha sido substituido por una interfaz gráfica a través de la consola de Python. Este cambio nos ha permitido redirigir esfuerzos a tareas más importantes que la presentación visual.

Como resultado final hemos desarrollado una demo para ver como funcionarían los distintos módulos de la aplicación y la comunicación entre ellos: ClimateFront. A continuación veremos en qué consiste, sin entrar en detalles técnicos, que serán abordados en los siguientes apartados referentes a la arquitectura del sistema.

ClimateFront

ClimateFront nace con el propósito de ver cómo funcionaría una aplicación como la propuesta y sentar las bases a nivel estructural en lo que se refiere a arquitectura y organización, subobjetivos del proyecto.

Al haber escogido MongoDB como sistema de persistencia vamos a trabajar con colecciones, el equivalente a tablas en SQL. Observando las necesidades del sistema hemos creado 3 colecciones: *product*, *route* y *plan*. Son las encargadas de guardar toda la información, aunque dentro de ellas encontramos distintas entidades, como las previsiones meteorológicas. La colección *product* guarda la información de los productos de la empresa, *route* contiene la información de las distintas rutas, y *plan* recoge los planes de las rutas planificadas junto con las previsiones meteorológicas usadas.

El módulo de OPC UA de la demo permite simular un servidor que actúa como controlador de entradas en un almacén y sincronizarse con él usando un cliente. Dicha sincronización consiste en propagar la información de las nuevas entradas de stock a la colección *product* de mongo. Por ejemplo: si el almacén registra la entrada de un paquete con 3 manzanas, la cantidad de manzanas guardada en mongo se incrementa en 3 unidades. Los productos de la colección *product* han sido creados en función de los productos del stock del servidor, y su identificador debe coincidir con el identificador del nodo en el servidor.

El servidor es accesible en la URL “opc.tcp://localhost:4840/stock”[11]. Tiene un nodo llamado “Stock” registrado en el namespace "http://climateFront.tfg.fib.upc". Del nodo “Stock”, a su vez, cuelgan tres nodos que contiene las variables: “Tomatoes”, “Bananas” y “Apples”. Una variable, dentro de contexto OPC UA, no es más que un nodo con ciertos atributos y comportamientos ya definidos. Cada variable representa la cantidad recibida en una nueva entrega, a través de la demo podemos simular entregas de cantidades aleatorias bajo demanda.

Para propagar los cambios a la colección *product* el cliente se suscribe a las variables y recibe una notificación cuando hay algún cambio en su valor. La notificación permite guardar la información en mongo acorde al servidor.

Para poder trabajar con las colecciones usamos el módulo de la API REST y operaciones CRUD (Create, Read Update, Delete). A través de la API podemos consultar todas las colecciones, crear rutas y planificarlas. Sin embargo, no podemos crear productos ni actualizar la cantidad directamente, ya que de esta tarea se encarga el módulo de OPC UA.

Al crear una ruta hay que indicar un conjunto de productos, y para cada uno de ellos se comprueba que exista en la colección de *product* a través del identificador y que la cantidad solicitada es inferior a la disponible. Si todo es correcto, se decrementa el stock de los productos acorde a la cantidad solicitada y se registra la ruta en la colección *route*. En caso de que la ruta fuese cancelada, el stock de los productos se restauraría.

Una ruta tiene tres estados posibles: pendiente (PENDING), planeada (PLANNED) y cancelada (CANCELED). Solo se podrán actualizar las rutas que estén en estado *PENDING*, que es el estado inicial. El plano de una ruta se crea de forma automática al actualizar el estado de la ruta de pendiente a planeada. El plano se programa en función de la estrategia de planificación de la ruta, especificada en la creación de esta. Una vez creado, se devuelve un objeto con el plan, los datos de la ruta y las previsiones climáticas que se han tenido en cuenta en el proceso. Este objeto es él que se almacena en la colección plan.

Actualmente solo tenemos la estrategia StochasticVRPMultiDepotStrategy disponible, que implementa el algoritmo de enrutamiento de vehículos de depósito múltiple (VRP multi-depot) con tiempos de viaje estocásticos, aunque en un futuro se podrán incorporar más. Antes de calcular el plan se solicitan las previsiones meteorológicas al web service de OpenWeatherMap.

La estrategia StochasticVRPMultiDepotStrategy simula calcular un plan siguiendo el algoritmo expuesto en apartados anteriores y con las previsiones climatológicas solicitadas previamente. No obstante, no se ha implementado el algoritmo en sí debido a la complejidad y no ser el objetivo del proyecto.

[11] Las aplicaciones OPC UA usan el puerto 4840 y el protocolo opc.tcp, localhost en mi caso por estar en el entorno de desarrollo.

Persistencia: MongoDB

MongoDB es una base de datos distribuida orientada a documentos. En lugar de guardar los datos en registros los guarda en documentos, estos se codifican en formato BSON, que es una representación binaria de JSON. Solo se garantiza atomicidad a nivel de documento, en mongo no existen transacciones.

Los documentos se asocian en objetos del tipo *collection*, y estas colecciones a su vez se agrupan por objetos del tipo *database*, lo cual permite definir aspectos comunes como usuarios y funciones sobre el conjunto. Dentro de una base de datos de mongo podemos encontrar más de un objeto del tipo *database*.

Diseño

A la hora de diseñar el modelado de los datos hay tener muy en cuenta qué tipo de consultas se realizarán para guardar la información acorde y optimizar el tiempo de respuesta. Si en un futuro se cambiasen las consultas lo más seguro es que se tuviese que cambiar el esquema, con lo que es muy importante tener claras las peticiones que nos van a llegar desde el principio.

Todo lo que respecta al diseño se puede simplificar en seguir una regla básica:

1 búsqueda/consulta para todo el conjunto de datos requerido

¿Cómo conseguimos esto?

El *embedding* es un factor clave que nos ayudará a conseguir este objetivo. Implica la desnormalización de los datos, almacenando dos (o más) documentos relacionados en uno único. Las operaciones sobre este documento final serán mucho menos costosas para el servidor.

Con el embedding se busca la independencia de los documentos ya que en mongo no existen las joins como tal. Para consultar datos relacionados en dos o más colecciones hay que hacer más de una consulta y montar la join en el código. Por eso, entre otras cosas, es importante saber qué consultas vamos a recibir desde el principio.

En general, se debe de emplear esta técnica cuando se tienen relaciones del tipo «contiene» entre entidades. En nuestro caso, esta situación se da con las rutas y los productos.

Respecto a la optimización de consultas, nuestra mejor arma será el uso de índices. Hay que indexar todos los campos por los que se realizarán búsquedas, ya que la diferencia entre realizar una consulta sobre campo con índice, y realizarla sin él, puede ser abismal.

Para mejorar la eficiencia de los índices, que se traducen en árboles B-Tree, es recomendable que estos tengan una cardinalidad alta. Cuantos más valores únicos tenga el campo, más alta será la cardinalidad y más eficiente será el índice.

Teniendo en cuenta todo esto, se han creado tres colecciones: ***product***, ***route*** y ***plan***, que contendrán la información de las distintas entidades con las que vamos a trabajar en la aplicación:

- **Product**: Productos del stock de la empresa.
- **Route**: Rutas creadas.
- **Strategy**: Estrategia a seguir para programar el plan de una ruta.
- **Plan**: Planes de las rutas planificadas.
- **LocationForecast**: Previsión meteorológica de una localidad durante un periodo de tiempo.
- **DayHourForecast**: Previsión meteorológica de una localidad en un día y hora concreto.

UML

UML (Fig. 5) las entidades que contendrán las colecciones de mongo.

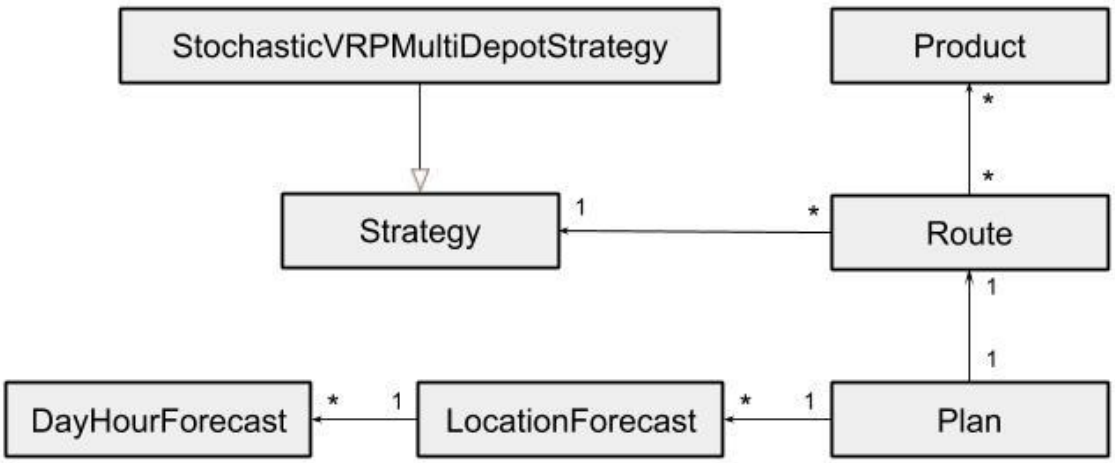

Fig. 5: UML del sistema

Colecciones

MongoDB asigna automáticamente un identificador único del tipo ObjectId como clave primaria a cada documento y lo llama "_id". En las posteriores explicaciones de las colecciones se ignora este identificador a la hora de comentar la estructura de un documento. En caso de hacer referencia al identificador del documento, estaremos aludiendo al atributo "id" definido por nosotros.

Se ha decidido que todos los campos sean del tipo String para evitar conflictos/complicaciones con los tipos de datos. El formato de las fechas es año + mes + día para poder realizar comparaciones con los métodos de la clase String y evitar el casteo a objetos del tipo Date.

En el "Anexo D: Ejemplos de documentos de las colecciones de MongoDB" encontramos un documento de ejemplo para cada colección.

Collection: product

Contiene la información de los productos de la empresa.

Esquema:

- **id**: **String**. Identificador del producto. Indexado.
- **name**: **String**. Nombre del producto. Indexado.
- **quantity**: **String**. Cantidad disponible. Indexado.

Los tres campos están indexados ya que se van a buscar productos por todos los atributos. No obstante, lo más probable es que el índice de "quantity" no sea muy eficiente debido a su poca cardinalidad, aunque casi siempre es preferible usar un índice poco eficiente a no usar índice alguno, a no ser que haya poco espacio.

Collection: route

Contiene la información de las rutas de la empresa.

Esquema:

- **id**: **String**. Identificador de la ruta. Indexado.
- **state**: **String**. Estado de la ruta. Indexado.
- **origin**: **String**. Ciudad origen. Indexado.
- **destiny**: **String**. Ciudad destino. Indexado.
- **departure**: **String**. Fecha inicio. Indexado.
- **arrival**: **String**. Fecha fin. Indexado.
- **products**: **JSON Array**. Productos a transportar.
- **strategy**: **String**. Estrategia de planificación de la ruta.

Se incluye la información de los productos de forma embebida para evitar tener que hacer una segunda consulta y join en el código, de esta forma devolvemos toda la información necesaria de golpe.

Se indexan todos los atributos excepto el campo "products" y "strategy". No nos interesa buscar ni por productos ni por tipo de estrategia. Al igual que con "quantity" en la colección "product", el índice del campo "state" no será muy eficiente. Aparte de los índices simples (un solo campo), se añaden dos índices compuestos: "origin – destiny" y "departure – arrival", buscando optimizar las consultas que impliquen ambos campos.

Collection: plan

Contiene la información de los planes de las rutas planificadas.

Esquema:

- **id**: **String**. Identificador del plan. Indexado.
- **route**: **JSON**. Ruta del plan. Campo interno "id" indexado.
- **plan**: **String**. Plan a seguir.
- **location_forecasts**: **JSON Array**. Previsiones meteorológicas para una localidad durante un periodo de tiempo.
  Esquema:
  - **latitude**: **String**. Latitud de la localidad.
  - **longitude**: **String**. Longitud de la localidad.
  - **country**: **String**. Nombre del país de la localidad.
  - **city**: **String**. Nombre de la localidad.
  - **timezone**: **String**. Zona horaria de la localidad.
  - **start_forecast**: **String**. Fecha y hora de inicio de la previsión.
  - **end_forecast**: **String**. Fecha y hora de fin de la previsión.
  - **day_hour_forecasts**: **JSON Array**. Previsiones meteorológicas para una localidad en un día y hora en concreto.
    Esquema:
    - **date**: **String**. Fecha de la previsión.
    - **hour**: **String**. Hora de la previsión.
    - **weather**: **String**. Tiempo.
    - **weather_description**: **String**. Descripción del tiempo.
    - **temperature**: **String**. Temperatura.
    - **temperature_min**: **String**. Temperatura mínima.
    - **temperature_max**: **String**. Temperatura máxima.
    - **pressure**: **String**. Presión atmosférica.
    - **humidity**: **String**. Humedad.
    - **wind_speed**: **String**. Velocidad del viento.
- **date_creation**: **String**. Fecha de creación del plan.
- **hour_creation**: **String**. Hora de creación del plan.

Como se puede ver en el esquema, tenemos varios elementos embebidos:

- **route**: En la mayoría de casos querremos también información de la ruta.
- **location_forecasts**: No se ha hecho una colección aparte para la entidad LocationForecast por la volatilidad de los datos y porque esta información por sí sola no tiene mucho sentido.
- **day_hour_forecasts**: Lo mismo que con la entidad LocationForecast.

Se indexan los campos "id" y "route.id" para las futuras búsquedas.

Arquitectura del sistema

Dos módulos principales: **src** y **scripts**. “scripts” contiene los scripts organizados por sistema de persistencia y versión. Actualmente solo tenemos la carpeta de “mongo” y la de la versión actual: “00.00.00”.

Dentro de “src” encontramos todo el código de la aplicación divido en cuatro módulos interconectados:

- **mongo_adapter**: Clases relacionadas con el acceso y recuperación de datos de MongoDB.
- **opcua_communication**: Clases relacionadas con OPC UA.
- **api_rest**: Clases relacionadas con el negocio y la API REST.
- **services**: Clases relacionadas con los servicios externos.

En el “Anexo 5: Proyecto Python” encontramos la foto completa del proyecto Python creado.

Consideraciones previas

Documentación de los módulos

En python existen atributos a nivel de clase y a nivel de instancia. La diferencia fundamental es que los atributos de clase son compartidos por todas las instancias de esa clase, mientras que los atributos de instancia son particulares para cada objeto. Si la clase tiene atributos a nivel de clase se especificará explícitamente, en caso contrario, siempre que hablemos de atributos nos estaremos refiriendo a los atributos de la instancia.

La documentación de cada clase incluye una breve descripción, sus atributos y sus métodos. Para los atributos se indica su nombre, su tipo y una descripción (para los atributos de clase también); lo mismo para los métodos, cambiando el tipo por tipo de retorno. Se usa el tipo “void” si la función no devuelve nada, aunque formalmente no exista ese tipo en Python.

Código python

Todo el código susceptible a fallo se engloba dentro un try – catch que logea un mensaje de error donde se indica la clase, función, motivo del fallo, y el mensaje propio de la excepción. El código está en inglés por convención, al igual que los formatos y nombres de paquetes/clases/métodos/ constantes /variables, sin embargo, puede ser que en algún sitio se haya ignorado alguna convención porque lo hemos visto oportuno.

No hemos programado tests por falta de tiempo, tampoco se han echado en falta gracias a las herramientas de testeo externas y el uso de buenas prácticas.

Servidor opcua.Server

Antes de hablar de las clases del módulo de *opcua_communication* conviene ver la estructura y valores por defecto de un servidor OPC UA creado con opcua.Server, partiendo del directorio ***Root*** encontramos tres directorios principales:

- ***Objects***: Donde se guardan los nodos, aquí encontramos el nodo "Server" que contiene toda la información del servidor y el estado actual. Ver Fig. 6.
- ***Types***: Agrupa por directorios los distintos tipos de Data, Event, Object, Reference y Variable. Si se crean nuevos tipos se añaden a la carpeta que toque. Ver Fig. 6.
- ***Views***: Vistas de información del servidor, vacío.

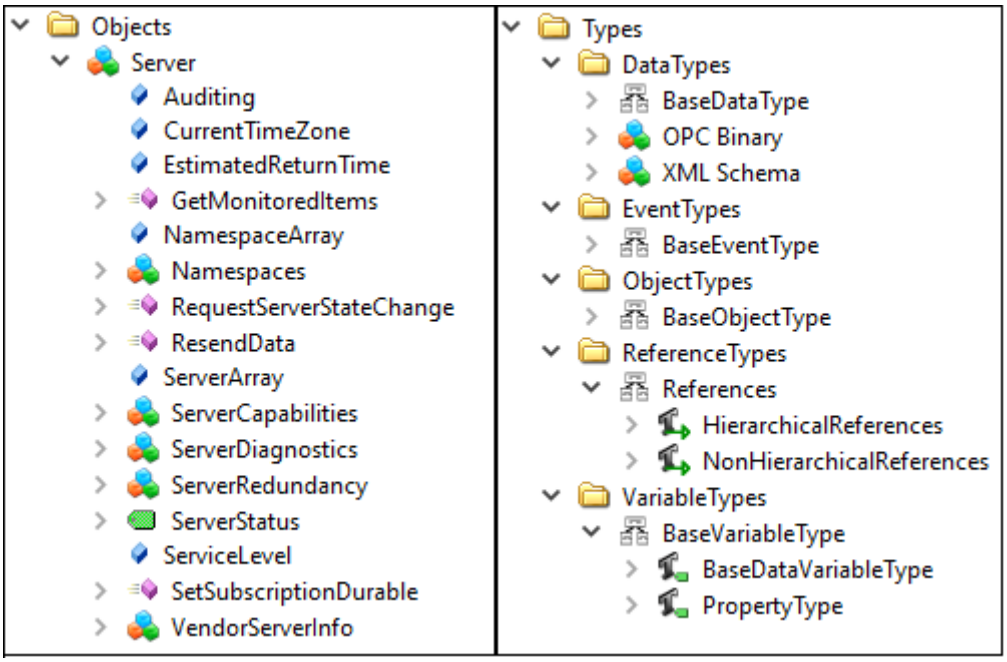

Fig. 6: Valores por defecto de los directorios "Objects" y "Types"

Cliente opcua.Client

El caso de uso típico y que veremos en este proyecto es la subscripción de un cliente opcua.Client a una variable del servidor. Estar suscrito a una variable permite recibir una notificación cada vez que se produce un cambio y poder realizar una acción como consecuencia. La notificación incluye bastante información, aunque el campo más importante es el que nos dice el nuevo valor que ha tomado la variable.

Para programar una acción de respuesta a una notificación lo único que tenemos que hacer es crear una clase que implemente la función *datachange_notification (self, node, val, data)*. La clase creada es la que pasaremos después a la suscripción (en la creación de esta) y la que definirá su comportamiento al recibir una notificación.

Ficheros comunes

config.py

Contiene la configuración de los distintos módulos a nivel de entorno de trabajo, actualmente solo está contemplado el entorno de desarrollo (DevelopmentConfig). La información se almacena en forma de constantes de clase, cada entorno se representa en un clase que hereda de la clase madre Config, que tiene la configuración común de todos los módulos y algunos valores por defecto.

El objeto que representa la configuración se crea en el fichero *__init__.py* del paquete *src*.

commons.py

Contiene las clases y constantes comunes de los distintos módulos, por el momento, los nombres de los campos de las colecciones de mongo y los posibles estados de una ruta.

ClimateFront.py

Interfaz gráfica y funciones para la demo de ClimateFront, en un futuro tendría que desaparecer. Por el momento permite realizar cuatro acciones, como vemos en la Fig. 7.

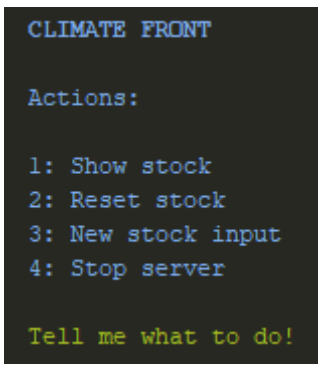

Fig. 7: Interfaz gráfica de ClimateFront

Módulo: mongo_adapter

Módulo que engloba las clases relacionadas con el acceso y recuperación de datos de MongoDB.

Class: MongoClientSingleton

Clase utilizada para hacer la clase __MongoClientSingleton *singleton*, la cual está definida dentro de esta.

Su objetivo es garantizar que solo habrá una instancia de la clase interna __MongoClientSingleton, asegurando así una única conexión abierta para realizar todas las peticiones.

Atributos de clase:

- **instance**: __MongoClientSingleton. Contiene una instancia de la clase interna __MongoClientSingleton, por defecto *None*.

Métodos:

- **def __new__ (cls)**: __MongoClientSingleton
  Si el atributo de clase "instance" es *None* le asigna una instancia del tipo __MongoClientSingleton, después devuelve el atributo de clase "instance".

Class: __MongoClientSingleton

Clase interna de MongoClientSingleton, se encarga de gestionar la conexión y hacer peticiones a mongo.

Atributos:

- **connection**: pymongo.MongoClient. Cliente de MongoDB.

Métodos:

- **def __init__ (self)**: void
  Inicializa la conexión, la URL de mongo la saca del objeto "config. MONGO_URL".
- **def getDatabase (self, databaseName)**: MongoDatabaseWrapper
  Crea y devuelve un objeto del tipo MongoDatabaseWrapper con la base de datos "databaseName".

- **def getCollection (self, collectionName, databaseName=config.MONGO_DATABASE)**: MongoCollectionWrapper
  Crea y devuelve un objeto del tipo MongoCollectionWrapper con la colección "collectionName" de la base de datos "databaseName". Si no se indica la base de datos se usa por defecto la del objeto "config".
- **def close (self)**: void
  Cierra la conexión "connection".

Class: MongoDatabaseWrapper

Clase encargada de gestionar las peticiones sobre una base de datos de mongo.

Atributos:

- **database**: pymongo.Database. Base de datos de mongo.

Métodos:

- **def __init__ (self, database)**: void
  Asigna la base de datos "database" al atributo "database".
- **def getCollection (self, collectionName)**: MongoCollectionWrapper
  Crea y devuelve un objeto del tipo MongoCollectionWrapper con la colección "collectionName" de la base de datos guardada en el atributo "database".

Class: MongoCollectionWrapper

Clase encargada de gestionar las operaciones sobre una colección de mongo.

Atributos:

- **collection**: pymongo.Collection. Colección de mongo.
- **collectionName**: String. Nombre de la colección en mongo.

Métodos:

- **def __init__ (self, collection, collectionName)**: void
  Asigna los parámetros a los atributos.
- **def insertOne (self, jsonDocument)**: pymongo.InsertOneResult
  Inserta el documento "jsonDocument" en la colección "collection". Devuelve un objeto con información de la operación de insertado.
- **def find (self, query)**: pymongo.MongoCursor
  Devuelve el resultado de la búsqueda "query" sobre la colección "collection". La consulta "query" ha de estar en formato JSON y seguir la sintaxis de mongo.

- **def updateOne (self, query, newValues)**: pymongo.UpdateResult
  Actualiza los documentos resultantes de la consulta "query" con los valores de "newValues", ambos campos han de estar en formato JSON y seguir la sintaxis de mongo. Devuelve un objeto con información de la operación de actualización.
- **def updateOneById (self, id, values)**: pymongo.UpdateResult
  Prepara los jsons de mongo para que solo tengas que indicar el id de los documentos que quieres actualizar ("id") y pasarle un diccionario de python (values) con los campos y valores a actualizar. Una vez están los jsons montados llama a la función *updateOne* descrita anteriormente.
- **def updateOneFieldById (self, id, field, newValue)**: pymongo.UpdateResult
  Prepara los jsons de mongo para que solo tengas que indicar el id de los documentos que quieres actualizar ("id"), el campo ("field") y el valor de ese campo ("newValue"). Una vez están los jsons montados llama a la función *updateOne* descrita anteriormente.

Módulo: opcua_communication

Módulo que engloba las clases relacionadas con el servidor/cliente OPC UA.

Dentro de este módulo encontramos el módulo *subscription*, contiene las clases usadas para crear subscripciones.

Class: ServerOPCUASimulation

Clase encargada de simular un servidor externo OPC UA.

Atributos:

- **server**: opcua.Server. Servidor OPC UA.
- **config**: Config. Configuración de la aplicación.
- **stockItems**: List [ocpua.NodeId]. Lista con las referencias de los nodos de las variables en el servidor.

Métodos:

- **def __init__ (self, config)**: void
  Inicializa el servidor con el endpoint y la política de seguridad de "config", después asigna el servidor creado y la configuración "config" a los atributos. El atributo "stockItems" se inicializa con una lista vacía.

- **def startSimulation (self)**: void
  Pone en marcha la simulación:
  - Registra el namespace “http://climateFront.tfg.fib.upc”.
  - Crea el nodo “Stock” en el directorio *Objects* y con el namespace registrado previamente.
  - Añade las variables al nodo “Stock” a través de la función *addStock*.
  - Se guarda las referencias de las variables obtenidas de la función *addStock* en al atributo “stockItems”.
  - Inicia el servidor.
- **def addStock (self, stock, namespaceIndex)**: List [ocpua.NodeId]
  Añade las variables “Tomatoes”, “Bananas” y “Apples” al nodo “Stock” con el namespace “namespaceIndex”. Se devuelve un objeto con las referencias a cada una de ellas.
- **def randomStockInput (self)**: void
  Resetea las variables del atributo “stockItems” en un cantidad aleatoria entre 0 y 10. Simula una nueva entrada de stock en el almacén.
- **stopServer (self)**: void
  Para el servidor.

Class: ClientOPCUA

Clase encargada de conectarse y hacer peticiones a un servidor opcua.Server.

Atributos:

- **client**: opcua.Client. Cliente OPC UA.

Métodos:

- **def __init__ (self, urlServer)**: void
  Inicializa el cliente y se conecta al servidor en la URL “urlServer”.
- **def getObjectsVar (self, nodeName, varName):** ocpua.NodeId
  Devuelve la referencia de la variable “varName” del nodo “nodeName” del directorio “Objects”.
- **def subscribeToVar (self, var):** opcua.Subscription
  Devuelve la subscripción creada con la clase SubscriptionHandler sobre la variable “var”.
- **def subscribeVarToMongoCollection (self, var, collectionWrapper):** opcua.Subscription
  Devuelve la subscripción creada con la clase SubscriptionMongoCollectionHandler y la colección “collectionWrapper.collection” sobre la variable “var”.

Class: SubscriptionHandler

Clase que define el comportamiento de una subscripción.

Métodos:

- **def datachange_notification (self, node, val, data):** void
  Muestra por pantalla y logea la información de la notificación recibida, donde el nodo es “node”, el valor es “val” y data (información detallada de la notificación) es “data”.

  Ejemplo:

```
Node: Node(NumericNodeId(ns=2;i=4)) Value: 1 Data:
 DataChangeNotification(<opcua.common.subscription.SubscriptionItemData
 object at 0x000001A3561D0288>, MonitoredItemNotification
 (ClientHandle:201, Value:DataValue(Value:Variant(val:1,type:VariantType
 .Int64), StatusCode:StatusCode(Good), SourceTimestamp:2020-03-30
 11:33:20.426562)))
```

Fig. 8: Ejemplo del log de una notificación usando la clase SubscriptionHandler

Class: SubscriptionMongoCollectionHandler

Clase que define el comportamiento de una subscripción ligada a una colección de mongo.

Atributos:

- **collectionWrapper**: MongoCollectionWrapper. Colección de mongo.

Métodos:

- **def __init__ (self, collectionWrapper)**: void
  Asigna el parámetro “collectionWrapper” al atributo “collectionWrapper”.
- **def datachange_notification (self, node, val, data):** void
  Realiza una acción u otra en función de la colección de mongo con la que estemos trabajando, que lo sabemos gracias al atributo “collectionWrapper.collectionName”.
  Si la colección es *product*, incrementamos el campo “quantity” del documento con el mismo “id” que el nodo de la notificación con el valor “val”.

Módulo: api_rest

Módulo que engloba las clases relacionadas con la API REST y el negocio.

Incluye tres módulos:

- **model**: Abarca las clases que definen las entidades con las que vamos a trabajar.
- **service**: Contiene la lógica de negocio de la aplicación.
- **controller**: Contiene el controlador de la API.

El fichero ***utils.py*** se encuentra al mismo nivel que los módulos especificados y contiene los métodos y constantes que se van a usar en todos ellos. Las funciones de casteo de documentos mongo a entidades, por ejemplo, están definidas aquí.

Módulo: controller

Contiene el controlador de la API, ***Controller.py***, donde se definen los endpoints de la aplicación (ver "Anexo 5: Endpoints API REST").

En caso de recibir una petición errónea devolvemos una BadRequest (código 400) donde se indica el motivo, "message", y un código de error personalizado, "status". Prefijo 4000 para los errores relacionados con los productos, 4001 para las rutas y 4002 para los planes.

Módulo: model

Subdividido en dos módulos:

- **entity**: Entidades de la aplicación.
  Toda entidad debe heredar de la clase Entity, definir la constante "ENTITY_NAME" e implementar el método *toJson (self)* para poder guardar los datos en mongo. Una clase para cada entidad, el nombre de la clase es el prefijo "Entity" más el nombre de la entidad. Los campos de mongo están mapeados contra los atributos de la instancia de la clase en la creación del objeto.
  Ejemplo: Para la entidad "Product" tenemos la clase EntityProduct con la función *__init__ (self, id, name, quantity)*.

- **planning_strategies**: Estrategias de planificación de las rutas.
  Toda estrategia debe heredar de la clase Strategy, definir la constante "STRATEGY_NAME" e implementar el método def *planIt (self, route)*. Hemos implementado el patrón *Strategy*.

Módulo: service

Contiene la lógica de negocio de ClimateFront:

- **ProductService.py**: Funciones relacionadas con los productos.
- **RouteService.py**: Funciones relacionadas con las rutas.
- **PlanService.py**: Funciones relacionadas con los planos.

Las funciones trabajan y devuelven, en caso de ser necesario, objetos del tipo Entity.

ProductService.py

Métodos:

- **def getProducts (filters = None)**: Array [EntityProduct]
  Devuelve el conjunto de productos que cumple los filtros especificados, por defecto *None*.
- **def checkProductsStock (products)**: (Boolean, Product, Integer)
  Para cada producto del conjunto "products" comprueba que exista y haya la cantidad solicitada.
  Si el producto no existe devuelve (*False*, producto inexistente, 0), si existe pero no hay la cantidad necesaria devuelve (*False*, producto, 1), y si todos los productos cumplen las condiciones devuelve (*True*, *None*, *None*).
- **def modifyProductsStock (products, operation)**: void
  Actualiza la cantidad de los productos en mongo acorde a la cantidad de los productos "products" y el tipo de operación indicado en "operation", que puede ser de incremento o decremento.

RouteService.py

Métodos:

- **def getRoutes (filters = None)**: Array [RouteEntity]
  Devuelve el conjunto de rutas que cumple los filtros especificados, por defecto *None*.
- **def addRoute (origin, destiny, departure, arrival, productsJson, strategy)**: None
  Decrementa los productos acorde a las cantidades indicadas en "productsJson" usando la función *modifyProductsStock* del fichero *ProductService.py*, después crea una nueva ruta con los parámetros como atributos y la guarda la colección *route* de mongo.
- **def checkEditRoute (route)**: Boolean
  Comprueba si la ruta "route" se puede editar. *True* si el estado es "PENDING", *False* en cualquier otro caso.
- **def updateRoute (originalRoute, fieldsToUpdate)**: EntityRoute
  Actualiza en mongo la ruta "originalRoute" acorde a los campos y valores del diccionario "fieldsToUpdate". Devuelve la ruta actualiza releída de mongo.
- **def cancelRoute (route)**: EntityRoute
  Pone el estado cancelado a la ruta "route" y restaura la cantidad de los productos acorde los productos de "route.products".

PlanService.py

Métodos:

- **def getPlans (filters = None)**: Array [EntityPlan]
  Devuelve el conjunto de planos que cumple los filtros especificados, por defecto None.
- **def addPlan (route)**: EntityPlan
  Genera un plan ejecutando el método *planIt* del objeto del atributo "route.strategy" y lo añade a la colección plan. Devuelve el plan creado.

Módulo: services

Clases relacionadas con los servicios externos. Cada servicio se representa en un módulo, por el momento solo tenemos el módulo **openWeatherMap**.

Servicio: openWeatherMap

Proporciona información meteorológica sobre una localidad. La respuesta está en formato JSON y contiene una lista de bloques de información meteorológica para una localidad (ver "Anexo F: OpenWeatherMap" para más detalle).

Se seleccionan los datos más relevantes y se encapsulan en un objeto del tipo EntityDayHourForecast. La agrupación de estos objetos junto con información de la localidad forma un objeto del tipo EntityLocationForecast.

Métodos:

- **def getCityId (codeIATACity):** String
  Devuelve el código de identificación de una localidad en OpenWeatherMap en función del código IATA.
- **def getCityForecast (codeIATACity, startDay=datetime.now ().strftime ("%Y%m%d"), days = 5, hours = 3)** : EntityLocationForecast
  Devuelve la previsión meteorológica de una localidad para los próximos cinco días en periodos de tres horas. En un futuro se podrá configurar el inicio de la previsión, el número de días y las horas del periodo. Por el momento nos adaptamos a la versión gratuita de cinco días y periodos de tres horas desde el día actual (valores por defecto de la función).

Validaciones

UaExpert

UaExpert es un cliente OPC UA multiplataforma programado en C ++, proporciona una interfaz gráfica muy potente que ha acelerado todo el proceso de validación

Lo hemos usado para testear el servidor OPC UA y para desarrollar el cliente OPC UA de la aplicación, nos ha permitido ver si se están propagando correctamente los valores del servidor cuando se actualiza. También nos ha servido para ver que identificadores asigna el servidor a las distintas variables creadas sin tener que averiguarlo por código.

UaExpert nos permite seleccionar distintos ítems dentro del servidor y mostrar información de su estado y evolución en tiempo-real, como vemos en la Fig. 9.

Node Id	Display Name	Value	Datatype	Source Timestamp	Statuscode
NS0|Numeric|2255	NamespaceArray	{'http://opcfo...	String	20:58:45.124	Good
NS0|Numeric|2256	ServerStatus	Double click t...	Extension...	20:58:45.123	Good
NS0|Numeric|2994	Auditing		Null	1:00:00.000	Good
NS2|Numeric|3	Bananas	11	Int64	20:58:45.190	Good
NS2|Numeric|4	Apples	13	Int64	20:58:45.190	Good
NS2|Numeric|2	Tomatoes	7	Int64	20:58:45.190	Good

Fig. 9: UaExpert: Panel visual "Data Acces View"

Insomnia

Cliente HTTP usado para testear los endpoints de la API REST, su simplicidad ha reducido prácticamente a cero el tiempo de aprendizaje de la herramienta.

Permite crear y lanzar requests HTTP personalizadas de una forma muy simple y cómoda gracias a la interfaz gráfica que ofrece.

Simulación: FlexSim

Hemos usado FlexSim para simular la planta de un almacén. Orden de ejecución del modelo:

1. Una fuente emite aleatoriamente cuatro tipos de cajas distintas diferenciadas por colores: rojo, verde, amarillo y azul. El tiempo de emisión es aleatorio siguiendo la función: *exponential (0, 10, getstream (current))*.
2. Las cajas entran a una cinta transportadora, las azules y rojas salen en la primera salida, las amarillas y verdes en la segunda.
3. A cada una de salidas de la cinta hay un operador que las transporta a un estante concreto de un *rack* en función de su color. La Fig. 10 ilustra los puntos 1, 2 y 3.
4. Un robot las mueve las cajas del estante a la cola correspondiente, una cola para cada tipo de caja.
5. Un transportista coge un palé y lo mueve hasta el combinador, este necesita un número concreto de cajas para montar un paquete. Estas cantidades son variables y están indicadas en la tabla "combiner_table", actualmente: una amarilla, tres azules, dos verdes y una roja. El transportista va llevando las cajas al combinador a medida que van apareciendo en las colas.
6. Una vez el combinador tiene el palé y las cajas necesarias monta un paquete que deja en la siguiente cola. En la Fig. 11 vemos elementos descritos en los puntos 4, 5, 6.
7. Un operador mueve los paquetes de la cola al procesador.
8. El procesador evalúa los paquetes durante un tiempo de 10 segundos.
   El procesador tiene una tasa de rotura definida por la función: *exponential (0, 1000, getstream (current))*, un vez averiado tarda un tiempo aleatorio a repararse definido por la función: *uniform (50, 100, getstream (current))*.
9. Después de la evaluación los paquetes se dejan en la salida. En la Fig. 12 se ve la parte final del modelo (puntos 7, 8 y 9).

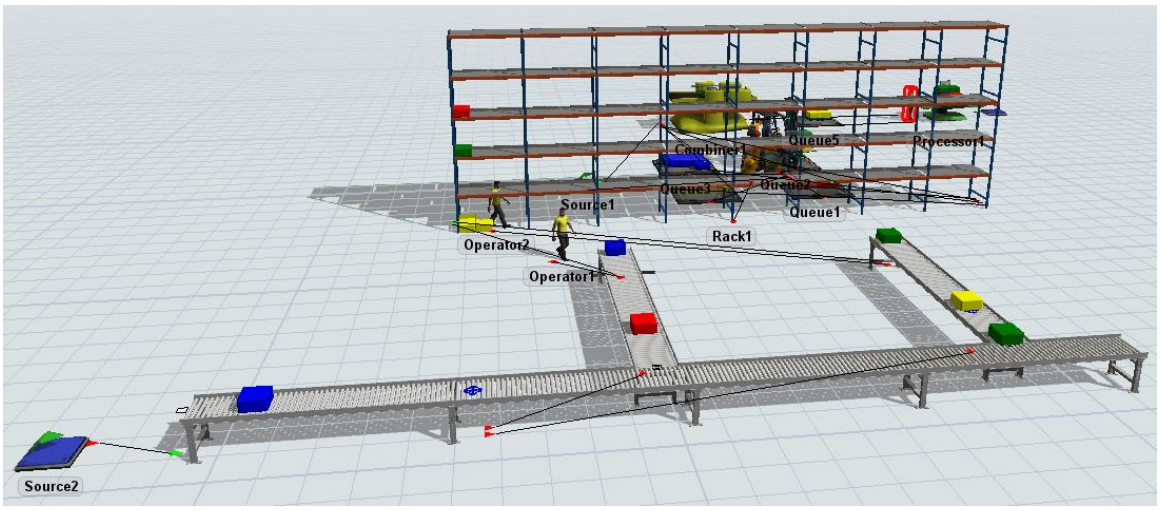

Fig. 10: Fuente, cinta transportada, operarios y rack del modelo FlexSim

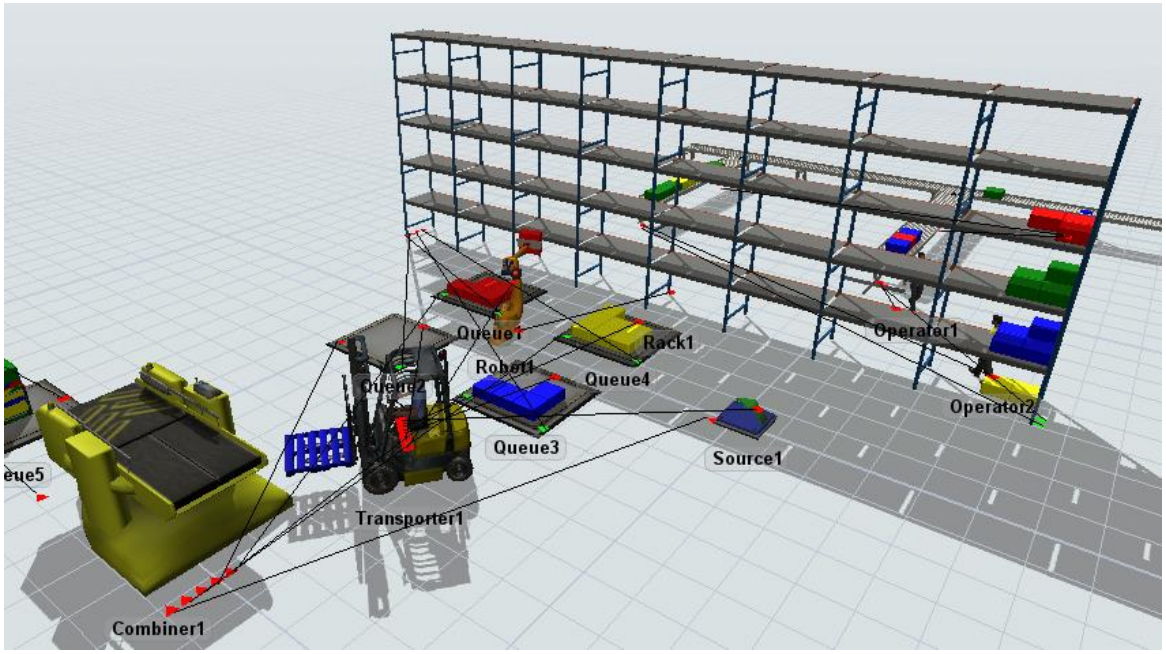

Fig. 11: Robot, transportista y combinador del modelo FlexSim

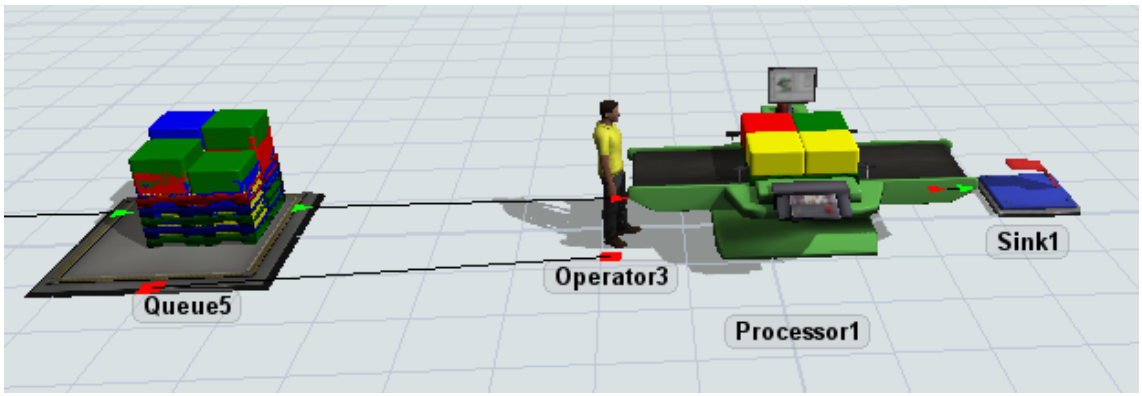

Fig. 12: Procesador y salida del modelo FlexSim

Un vez tenemos el modelo montado ya se puede empezar a usar el módulo de estadísticas, la mejor herramienta de FlexSim. Los distintos gráficos que ofrece el software nos permitirán ver los puntos flojos de sistema y observar que sucede si modificamos algún parámetro o introducimos más elementos al modelo. A continuación mostramos distintos gráficos a modo de ejemplo (Fig. 13, Fig. 14, Fig. 15).

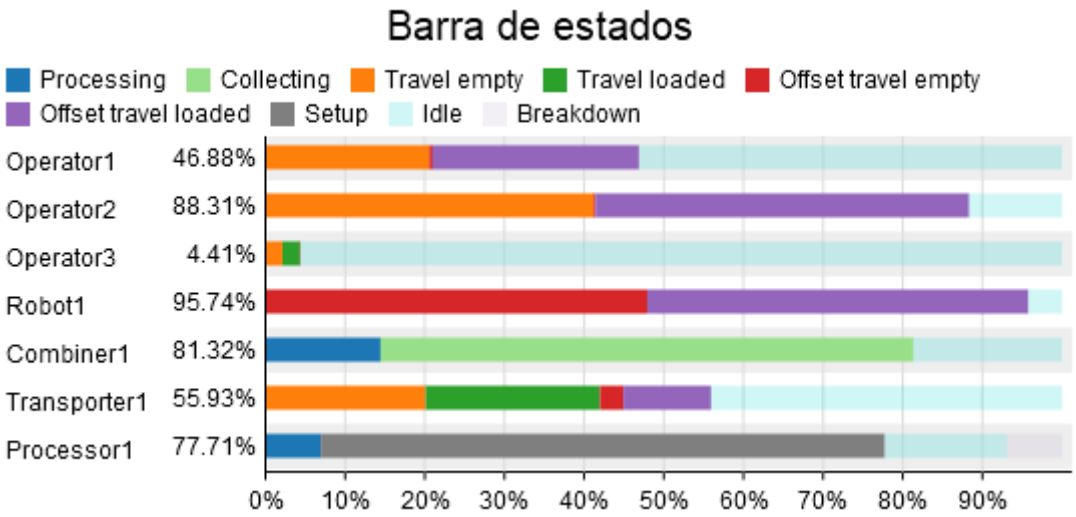

Fig. 13: FlexSim: Tanto por ciento del tiempo dedicado a cada estado para cada elemento del sistema transcurridos 60 min

Salida de elementos

Object	Throughput
Combiner1	52
Processor1	25

Fig. 14: FlexSim: Número de elementos creados por el combinador y evaluados por el procesador transcurridos 60 min

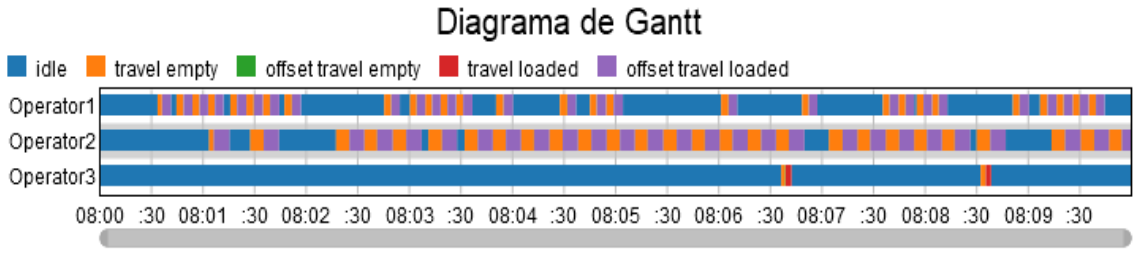

Fig. 15: FlexSim: Diagrama de Gantt para los tres operarios del sistema transcurridos 10 minutos

Usando estos gráficos podríamos estudiar introducir más operarios/robots y ver qué sucede y cómo afecta a la producción, por ejemplo, pero esto ya queda fuera de alcance de este proyecto e incluso daría para otro TFG.

Conclusiones

Reducir el desperdicio de alimento; el objetivo final y motivación principal de este proyecto.

Mucha gente se centra en acciones para combatir el cambio climático, una causa muy noble, pero también hace falta gente que adapte los procesos actuales a las inevitables consecuencias que provocará y está provocando el calentamiento global. ClimateFront pretende mejorar la distribución y conservación de las mercancías optimizando la logística que hay detrás, a día de hoy carente de información vital como las previsiones meteorológicas de la ruta o el control ininterrumpido de los productos.

Como hemos visto anteriormente, la mayoría de las aplicaciones vigentes son pobres y están basadas en sistemas cerrados y obsoletos, hace falta un nuevo enfoque. Los sistemas deben tener en cuenta múltiples canales de información (interna y externa) y saber responder de forma automática a ciertos estímulos.

Nuestra apuesta por OPC UA, servicios web y nuevos algoritmos de distribución no solo persigue mejorar la oferta actual, sino hacer frente a un futuro incierto en lo que respecta al clima de los distintos países. Con el diseño de nuestra demo de ClimateFront hemos expuesto la arquitectura de una aplicación de logística adaptada a los nuevos tiempos y capaz de incorporar información interna y externa con facilidad y seguridad.

Nos ha faltado recrear el ecosistema IoT para el control total de los productos, pero nuestra aproximación de los que sería un servidor OPC UA y sus formas de proceder marcan el camino a seguir en un futuro. El sistema de persistencia NoSQL, MongoDB, también está preparado y pensado para manejar los grandes flujos de información procedentes de los distintos dispositivos.

Esperamos que nuestro proyecto sirva para que alguien continúe con la tarea aquí empezada y podamos ver algún día cómo una aplicación rebasa a la competencia y planta cara a los envites meteorológicos: ClimateFront, el héroe logístico que necesitamos.

Referencias

AGM – Larraioz Elektronika., 2016. OPC: Desde el clásico al nuevo OPC-UA. En: [en línea]. [consulta: 6 octubre 2019]. Disponible en: https://larraioz.com/articulos/opc-desde-el-clasico-al-nuevo-opc-ua.

Alsina, A., 2018. Beneficios del IoT o Internet of Things aplicado a la logística. En: [en línea]. [consulta: 6 octubre 2019]. Disponible en: https://www.iebschool.com/blog/internet-of-things-sector-logistica/.

Estrada-Moreno, A. et al., 2018. Distribution planning in a weather-dependent scenario with stochastic travel times: A simheuristic approach. En: [en línea]. p. 12. Disponible en: https://www.informs-sim.org/wsc18papers/includes/files/265.pdf.

Fonseca i Casas, P.; Fonseca i Casas, A.; Garrido-Soriano, N.; Casanovas, J., 2014. Formal simulation model to optimize building sustainability. En: *Advances in Engineering Software*. Vol. 69, p. 62–74.

García, J.M.B., 2015. ¿Qué son los web services y qué tecnología usar en su desarrollo? En: [en línea]. [consulta: 6 octubre 2019]. Disponible en: https://www.arsys.es/blog/programacion/diseno-web/web-services-desarrollo/.

Gracia, M., 2019. IoT - Internet Of Things. En: [en línea]. [consulta: 6 octubre 2019]. Disponible en: https://www2.deloitte.com/es/es/pages/technology/articles/IoT-internet-of-things.html.

Lázaro, D., 2018. Introducción a los Web Services. En: [en línea]. [consulta: 6 octubre 2019]. Disponible en: https://diego.com.es/introduccion-a-los-web-services.

Leiva, J.; Fonseca i Casas, P.; Ocana, J., 2018. Modeling anesthesia and pavilion surgical units in a Chilean hospital with Specification and Description Language. En: *Simulation*. Vol. 89, p. 1020–1035.

OPC Fundation., 2017. OPC UA middleware enables highly flexible food production. En: [en línea]. Disponible en: http://opcfoundation.cn/resources/case-studies/OPC-UA-SuccessStory-Food-Beverage-Weber-v1.pdf.

Wikipedia., 2018. Wikipedia: Branding. En: [en línea]. [consulta: 6 octubre 2019]. Disponible en: https://es.wikipedia.org/wiki/Branding.

Wikipedia., 2019. Wikipedia: Modelo de Objetos de Componentes Distribuidos. En: [en línea]. [consulta: 6 octubre 2019]. Disponible en: https://es.wikipedia.org/wiki/Modelo_de_Objetos_de_Componentes_Distribuidos.

Anexos

Anexo A: OPC

OPC utiliza un planteamiento cliente-servidor para el intercambio de información: un servidor OPC encapsula la fuente de información de un proceso y la hace disponible a través de su interfaz.

Un cliente OPC se conecta a un servidor OPC para leer y consumir la información ofrecida, las aplicaciones que consumen y ofrecen datos pueden ser, a la vez, cliente y servidor.

Las interfaces clásicas de OPC están basadas en la tecnología COM y DCOM de Microsoft. La ventaja de este planteamiento era la reducción del trabajo en la especificación de la definición de diferentes APIs para diferentes necesidades especializadas al no tener que definir un protocolo de red o un mecanismo para comunicaciones entre procesos. COM y DCOM ofrecen a un cliente un mecanismo transparente para llamar a métodos de un objeto COM en un servidor que se está ejecutando, en el mismo proceso, en otro proceso o en otro nodo de red. Esta ventaja fue importante para el éxito de OPC. (AGM – Larraioz Elektronika 2016)

De acuerdo con las diferentes necesidades existen tres principales especificaciones OPC:

- Acceso a Datos (OPC-DA): Acceso a datos de proceso actuales
- Alarmas y Eventos (OPC-A&E): Información basada en eventos, incluyendo reconocimiento de alarmas
- Acceso a Datos Históricos (OPC-HDA): Acceso a datos históricos archivados.

La Fig. 16 ilustra el caso de uso típico:

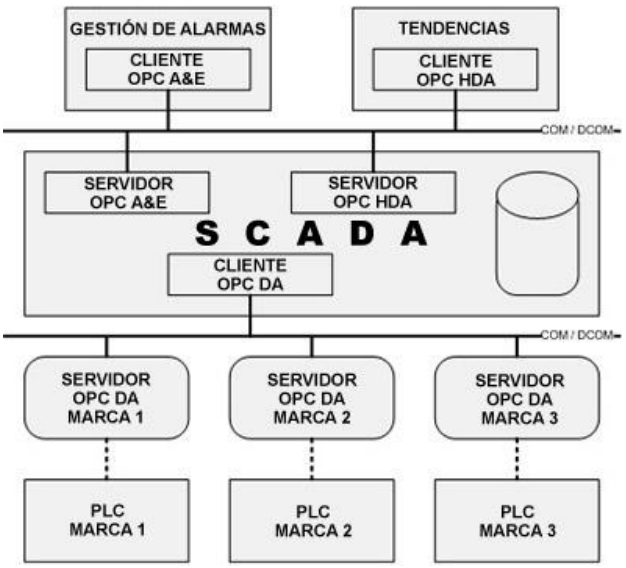

Fig. 16: Caso de uso típico en Clientes y Servidores OPC[12]

[12] Fuente: https://larraioz.com/articulos/opc-desde-el-clasico-al-nuevo-opc-ua

Esquema de comunicación entre aplicaciones antes (Fig. 17) y después (Fig. 18) de usar los estándares OPC para el intercambio de información:

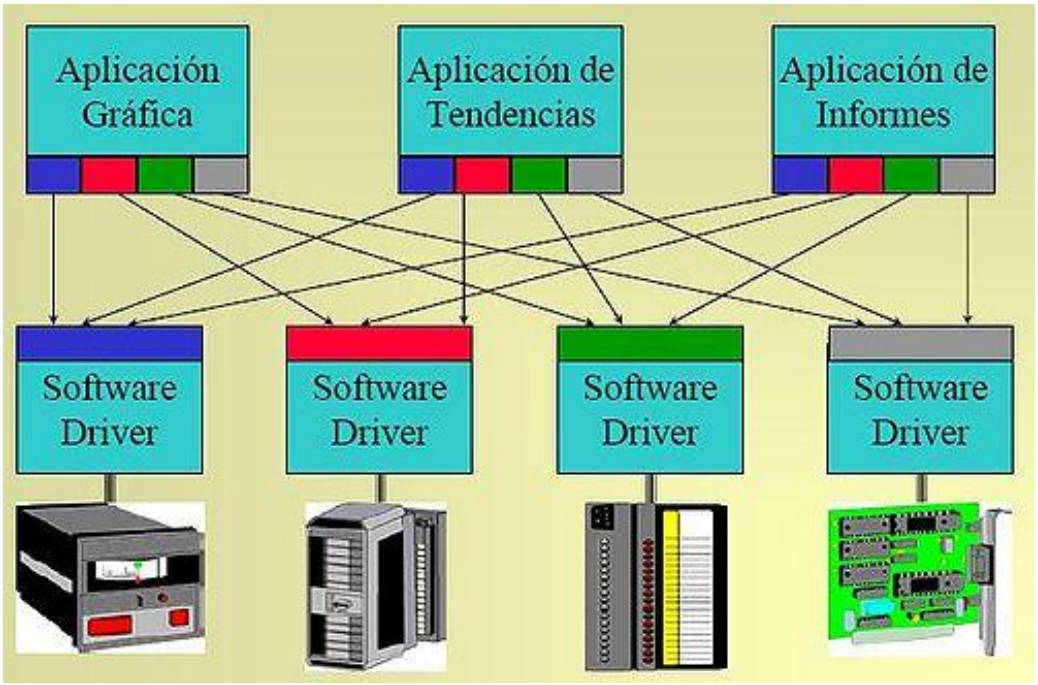

Fig. 17: Comunicación sin usar estándares OPC[13]

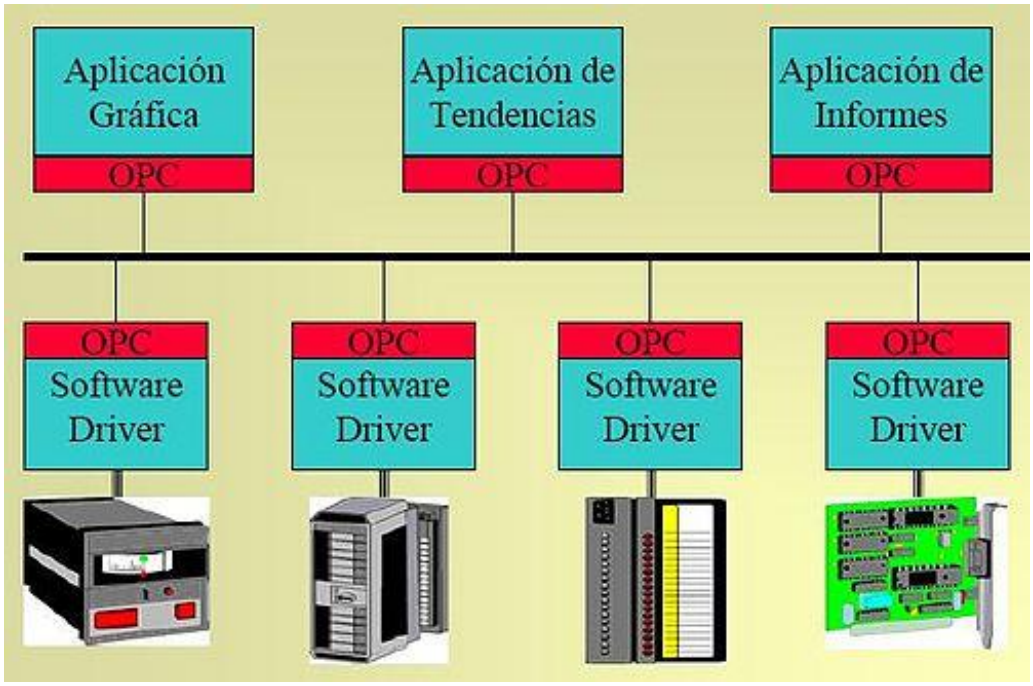

Fig. 18: Comunicación mediante estándares OPC[14]

[13] Fuente: https://es.wikipedia.org/wiki/OPC
[14] Fuente: https://es.wikipedia.org/wiki/OPC

Anexo B: VRP de múltiples depósitos con tiempos de viaje estocásticos, aproximación heurística

Extracto traducido del artículo (Estrada-Moreno et al. 2018):

En el escenario de un VRP de múltiples depósitos dependiente del clima, se supone que los tiempos de viaje son variables aleatorias que siguen distribuciones de probabilidad específicas. Por lo tanto, dejemos que G = {V, E} sea un gráfico no dirigido, donde V = Vf U Vc es el conjunto de vértices que incluye los depósitos o instalaciones (Vf) y los clientes (Vc), y E = {{u, v}: u, v ∈ Vc, u != v} U {{u, v}: u ∈ Vc, v ∈ Vf} es el conjunto de bordes que conecta los vértices en V. Cada cliente i ∈ Vc tiene una demanda positiva di > 0 que tiene que ser atendida. Cada depósito f ∈ Vf tiene asignado un número máximo de vehículos, mf, y no tiene demanda. Se supone que todos los vehículos tienen la misma capacidad Q, con Q >> max {di}.

Por lo tanto, la capacidad de servicio de un depósito está limitada por mf · Q. Viajar por cada arista en E tiene asociado un coste basado en el tiempo de viaje Tij = Tji > 0, que es una variable aleatoria. La parametrización de la distribución de probabilidad (log-normal) en la que se basan los costos de los viajes depende de las condiciones climáticas.

Una solución a este problema es un conjunto de mapas de clientes a instalaciones y los planes relacionados de rutas de ida y vuelta que salen de cada depósito y cubren todas las demandas de los clientes al tiempo que satisface todas las limitaciones de capacidad (tanto las relacionadas con los depósitos como las relacionados con los vehículos). Como en la mayoría de los VRP, se supone que cada cliente solo puede ser visitado una vez por un solo vehículo.

El objetivo principal en este caso es encontrar una solución factible que minimice los costos totales de tiempo de viaje esperados, mientras satisface las demandas del cliente y las limitaciones de capacidad. Incluso en su versión determinista, este problema representa un desafío ya que integra un problema de asignación combinatoria, en el que cada cliente está asignado a una instalación, con varios VRP, uno por instalación.

Al incluir también el componente estocástico, el problema se vuelve aún más difícil de resolver, lo que justifica el uso de un enfoque de optimización de simulación como el presentado en este documento. La Fig. 19 muestra una posible solución para una instancia particular.

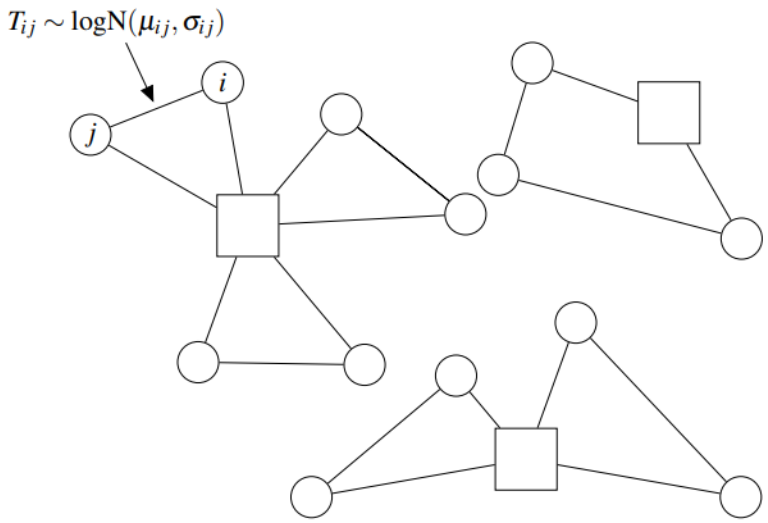

Fig. 19: Posible solución para VRP de múltiples depósitos con tres depósitos representados por cuadrados y varios clientes representados por círculos. Los tiempos de viaje se modelan con distribuciones log-normales

Nuestro enfoque se basa en dos hechos: (i) el VRP de depósito múltiple con tiempos de viaje estocásticos puede considerarse una generalización del VRP de depósito múltiple, es decir, el último puede verse como un caso particular del primero siempre que las demandas aleatorias tengan varianza cero; y (ii) a pesar del hecho de que la versión estocástica nunca se ha estudiado antes, existen algoritmos eficientes para resolver la versión determinista.

Las ideas generales detrás de nuestro enfoque, que se basa en un marco metaheurístico, se describen a continuación.

Inicialmente, dada una instancia del VRP de depósito múltiple con tiempos de viaje estocásticos, se transforma en una instancia determinista al reemplazar cada tiempo de viaje aleatorio por su valor esperado. Luego se obtiene un conjunto de soluciones de alta calidad para la versión determinista mediante el uso de un algoritmo metaheurístico eficiente que combina aleatorización sesgada con búsqueda local iterada. Mientras se realiza la búsqueda, la simulación de Monte Carlo se emplea para evaluar el rendimiento de estas soluciones prometedoras para la versión estocástica.

Definimos la mejor solución como la que tiene el menor costo total esperado basado en el tiempo de viaje. Esta evaluación se lleva a cabo de acuerdo con los siguientes pasos: (i) ejecute cientos de ejecuciones (hasta que se alcancen los criterios de detención), donde cada ejecución implica la generación de valores aleatorios para cada tiempo de viaje estocástico de acuerdo con la distribución de probabilidad asociada, que en cada turno dependerá de las condiciones climáticas específicas;

(ii) evaluar el rendimiento de cada solución estimando el costo total esperado basado en el tiempo como el promedio de los costos totales basados en los tiempos obtenidos al final de cada ejecución; y (iii) utilice la retroalimentación de simulación para guiar mejor el proceso de búsqueda dentro de la metaheurística.

La Fig. 20 muestra el diagrama de flujo del algoritmo completo. En la primera etapa del algoritmo, se propone una búsqueda local iterada (ILS) para la generación de nuevos mapas. Dado que es crítico evaluar tantos mapas de asignación como sea posible en el tiempo de cómputo disponible, cada vez que se genera un mapa, su costo de enrutamiento determinista se estima utilizando la conocida heurística de ahorro. Cada solución considerada prometedora desde el punto de vista determinista pasa a un proceso de simulación rápido (es decir, bajo número de repeticiones) para estimar su costo estocástico.

Más adelante se considera si esta solución se clasifica como solución de élite teniendo en cuenta su costo estocástico. Este proceso se repite hasta que se alcanza el criterio de detención. Una vez que se realiza la fase de "Interacción entre la búsqueda dirigida por metaheurística y la simulación", se realiza un paso intensivo de refinamiento. Para cada uno de los mapas de élite obtenidos en la primera etapa, se aplica un algoritmo de enrutamiento más intenso. Por supuesto, la cantidad de mapas de élite a considerar dependerá del tiempo disponible y los recursos informáticos.

Finalmente, se realiza un análisis de riesgo y confiabilidad sobre las soluciones de élite obtenidas en esta segunda etapa.

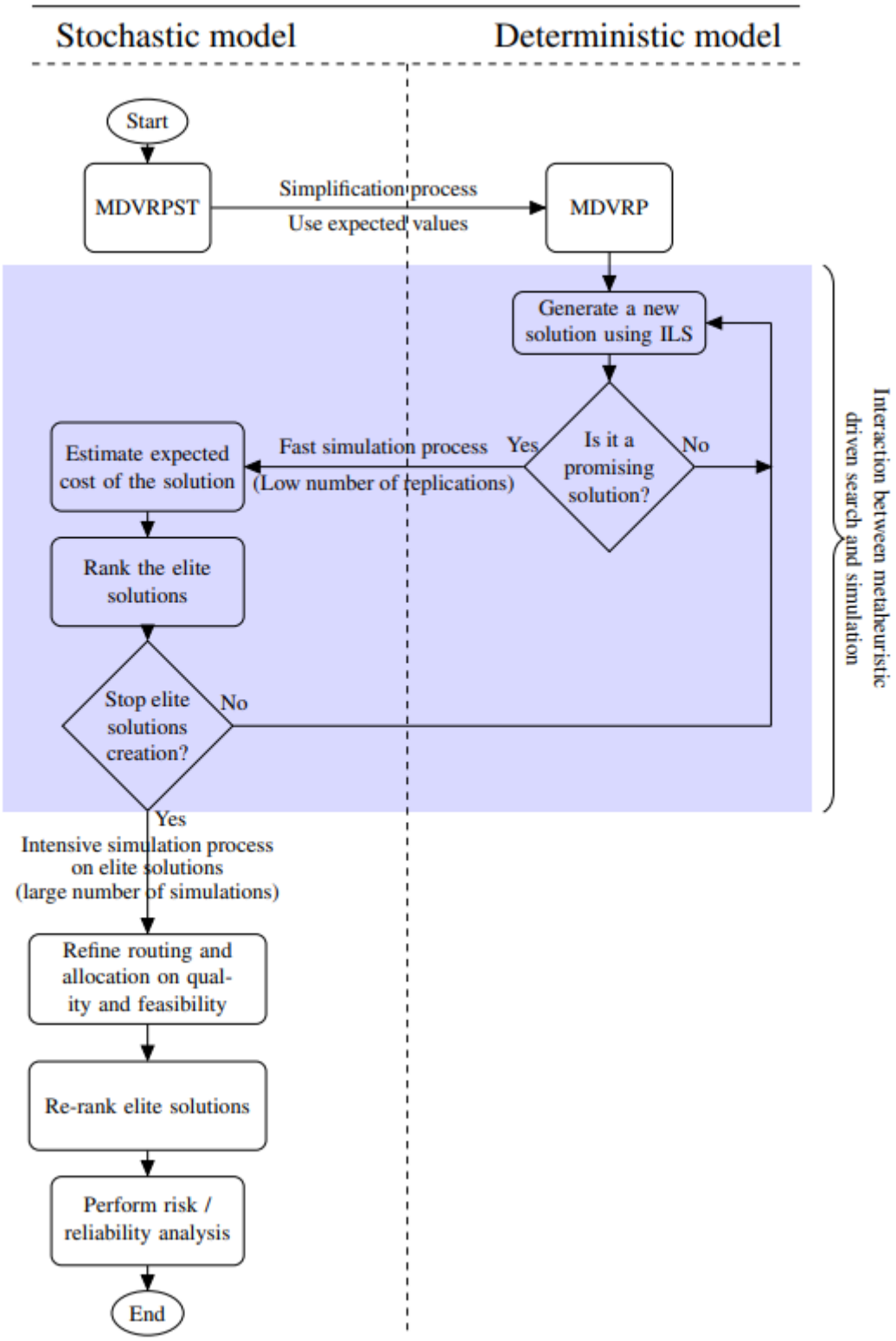

Fig. 20: Diagrama de flujo de la propuesta simheurística

Anexo C: El caso irlandés

Extracto traducido del artículo (Estrada-Moreno et al. 2018):

En el caso de Irlanda, se espera que el cambio climático aumente notablemente la frecuencia de eventos de fuertes precipitaciones durante el invierno, hasta un significativo 20% durante este siglo. Esto podría empeorar los costos y retrasos existentes debido a eventos relacionados con él. Por ejemplo, en febrero de 2018, 5 días de interrupciones del tráfico relacionadas con la nieve en todo el país causaron pérdidas para las empresas de transporte estimadas en más de 160 millones de euros. Como consecuencia, la mayoría de las tiendas y minoristas tuvieron que cerrar debido a los problemas con el suministro de bienes.

Anexo D: Ejemplos de documentos de las colecciones de MongoDB

Product


```
{
    "_id" : ObjectId("5e7789f8bb312aaa895b0ecd"),
    "id" : "2",
    "name" : "tomatoe",
    "quantity" : "0"
}
```


Fig. 21: Ejemplo de documento de la colección *product*

Route


```
{
    "_id" : ObjectId("5e89eb494bd6987e58105453"),
    "id" : "1586096969471052400",
    "state" : "PLANNED",
    "origin" : "BCN",
    "destiny" : "LON",
    "departure" : "20190115",
    "arrival" : "20190125",
    "products" : [
        {
            "id" : "2",
            "name" : "tomatoe",
            "quantity" : "5"
        },
        {
            "id" : "3",
            "name" : "banana",
            "quantity" : "2"
        }
    ],
    "strategy" : "StochasticVRPMultiDepotStrategy"
}
```


Fig. 22: Ejemplo de documento de la colección *route*

Plan


```
{
    "_id" : ObjectId("5e8c2cac13c4f8aae553e8ce"),
    "id" : "1586244780223713900",
    "route" : {
        "id" : "1586096969471052400",
        "state" : "PLANNED",
        "origin" : "BCN",
        "destiny" : "LON",
        "departure" : "20190115",
        "arrival" : "20190125",
        "products" : [
            {
                "id" : "2",
                "name" : "tomatoe",
                "quantity" : "5"
            }
        ],
        "strategy" : "StochasticVRPMultiDepotStrategy"
    },
    "plan" : "This is a fancy plan",
    "location_forecasts" : [
        {
            "latitude" : "41.3888",
            "longitude" : "2.159",
            "country" : "ES",
            "city" : "Barcelona",
            "timezone" : "7200",
            "start_forecast" : "20200407 09:00",
            "end_forecast" : "20200412 06:00",
            "day_hour_forecasts" : [
                {
                    "date" : "20200407",
                    "hour" : "09:00",
                    "weather" : "Clouds",
                    "weather_description" : "overcast clouds",
                    "temperature" : "14.18",
                    "temperature_min" : "14.18",
                    "temperature_max" : "15.70",
                    "pressure" : "1027",
                    "humidity" : "56",
                    "wind_speed" : "1.4"
                }
            ]
        }
    ],
    "date_creation" : "20200407",
    "hour_creation" : "093300"
}
```


Fig. 23: Ejemplo de documento de la colección *plan*

Anexo E: Arquitectura del sistema

Proyecto Python

```
ClimateFront/
    scripts/
        mongo/
            00.00.00/
                00_add_product.js
                01_add_route.js
                02_add_product_indexes.js
                03_add_route_indexes.js
                04_add_plan.js
                05_add_plan_indexes.js
    src/
        api_rest/
            controller/
                __init__.py
                Controller.py
            model/
                entity/
                    __init__.py
                    Entity.py
                    EntityDayHourForecast.py
                    EntityLocationForecast.py
                    EntityPlan.py
                    EntityProduct.py
                    EntityRoute.py
                planning_strategies/
                    __init__.py
                    StochasticVRPMultiDepotStrategy.py
                    Strategy.py
            service/
                __init__.py
                PlanService.py
                ProductService.py
                RouteService.py
            __init__.py
            utils.py
        mongo_adapter/
            __init__.py
            MongoClientSingleton.py
            MongoCollectionWrapper.py
            MongoDatabaseWrapper.py
        opcua_communication/
            subscription/
                __init__.py
                SubscriptionHandler.py
                SubscriptionMongoCollectionHandler.py
            __init__.py
            ClientOPCUA.py
            ServerOPCUASimulation.py
        services/
            openWeatherMap/
                __init__.py
                OpenWeatherMap.py
            __init__.py
        __init__.py
        ClimateFront.py
        commons.py
        config.py
```

Fig. 24: Estructura del proyecto Python

Endpoints API REST

Los métodos han sido implementados bajo demanda, solo están disponibles las operaciones que necesita la demo.

La columna “Status” indica los códigos de error que puede devolver el método en la BadRequest.

Product

Tabla 8: Endpoints de Product

Método	Ruta	Descripción	Status
GET	/products	Devuelve todos los productos.	400001
GET	/products/{product_id}	Busca un producto por el campo “id”.	400002

Route

Tabla 9: Endpoints de Route

Método	Ruta	Descripción	Status
GET	/routes	Devuelve todas las rutas.	400101
GET	/routes /{route_id}	Busca una ruta por el campo “id”.	400102
PUT	/routes /{route_id}	Actualiza una ruta.	400105, 400106
POST	/routes	Crea una ruta.	400103, 400104
DELETE	/routes /{route_id}	Cancela una ruta.	400105

Plan

Tabla 10: Endpoints de Plan

Método	Ruta	Descripción	Status
GET	/plans	Devuelve todos los planes.	-
GET	/plans/{plan_id}	Busca un plan por el campo “id”.	400201

Anexo F: OpenWeatherMap

Para obtener la información tenemos que llamar a la API de OpenWeatherMap con nuestra llave (appid) y el identificador de la localidad (id). El *appid* que nos lo dan al registrarnos y el identificador de la localidad los sacamos del fichero "city.list.json" descargable en la misma web[15].

La respuesta incluye campos relativos a HTTP, una lista con bloques de información meteorológica y características sobre la localidad consultada.

Estructura:

{*campos HTTP*, '**list'**: [{*bloques de información meteorológica*}], '**city'**:{ *características de la localidad*}}

Ejemplo con la localidad de Londres, GB.

Llamada a la API:

https://api.openweathermap.org/data/2.5/forecast?**appid**=fa5363235e9d90bef24be9a929e7f9c7&**id**=2643743

Campos HTTP:

'cod': '200', **'message'**: 0, **'cnt'**: 40

Ejemplo de un objeto del Array **'list'**:

{**'dt'**: 1585818000, **'main'**: {**'temp'**: 281.14, **'feels_like'**: 277.81, **'temp_min'**: 280.45, **'temp_max'**: 281.14, **'pressure'**: 1016, **'sea_level'**: 1016, **'grnd_level'**: 1013, **'humidity'**: 72, **'temp_kf'**: 0.69}, **'weather'**: [{**'id'**: 804, **'main'**: 'Clouds', **'description'**: 'overcast clouds', **'icon'**: '04d'}], **'clouds'**: {**'all'**: 100}, **'wind'**: {**'speed'**: 2.68, **'deg'**: 269}, **'sys'**: {**'pod'**: 'd'}, **'dt_txt'**: '2020-04-02 09:00:00'}

Campos del objeto **'city'**:

'id': 2643743, **'name'**: 'London', **'coord'**: {**'lat'**: 51.5085, **'lon'**: -0.1257}, **'country'**: 'GB', **'population'**: 1000000, **'timezone'**: 3600, **'sunrise'**: 1585805556, **'sunset'**: 1585852499}

[15] https://openweathermap.org/current